HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Daniel Schindler

Entwicklung eines Webshops mit Zcash-Zahlungsfunktionalität

2017

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Entwicklung eines Webshops mit Zcash-Zahlungsfunktionalität

Autor:
Daniel Schindler

Studiengang:
Angewandte Informatik: IT-Sicherheit

Seminargruppe:
IF13wI-B

Erstprüfer:
Herr Prof. Dr.-Ing. Andreas Ittner

Zweitprüfer:
Herr Prof. Dr.-Ing. Wilfried Schubert

Mittweida, 2017

Bibliografische Angaben

Schindler, Daniel: Entwicklung eines Webshops mit Zcash-Zahlungsfunktionalität, 49 Seiten, 13 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2017

Referat

Die Arbeit beschreibt die Entwicklung eines Webshops. Der Webshop ermöglicht dem Nutzer, Artikel mit der Kryptowährung Zcash zu kaufen. Dafür werden Front- und Back-End des Webshops implementiert. Das Back-End enthält Funktionen zur Kaufabwicklung mit Zcash.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Vorwort

Seit es Währungen gibt, gibt es Menschen die diese missbrauchen. Das Ausstellen von Scheinen ermöglicht das Erstellen von „scheinbaren“ Werten aus dem Nichts. Diese Schwachstelle der Fiat-Währungen wird im großen Stil missbraucht. Über Bankensysteme werden Wohlstand und Armut auf der ganzen Welt gesteuert. In den Köpfen der Menschen ist das Überleben ohne Geld längst nicht mehr möglich. Als würden sie es essen und atmen. Gier, Schulden, oder Angst vor einem geringerem Kontostand lässt die Menschen tatsächlich für mehr Arbeit demonstrieren. Das gilt heute als normal. Medien erklären den Menschen, jedes Land hätte einen großen Geldtopf mit Schulden, die man begleichen müsse. Einige unabhängiger Personen berechnen die Schulden jedoch höher, als die Summe des Besitzes aller Menschen. Würde man alle Schulden begleichen, so gäbe es kein Geld mehr und die Länder wären noch immer verschuldet. Kryptowährungen können diesen weltweiten Betrug stoppen. Die meisten dieser Geldsysteme gewährleisten ein gesundes Wachstum und eine maximale Geldmenge. Einige ermöglichen vollkommen anonyme Transaktionen. Die Akzeptanz von Kryptowährungen zu erhöhen, birgt viele Hürden. Das größte Problem ist eine fehlende Möglichkeit, mit diesen Währungen einzukaufen. Meine Arbeit zeigt die technische Umsetzung einer Lösung dieses Problems. Die Akzeptanz von Kryptowährungen und deren hohe Volatilität kann mit dieser Lösung jedoch nicht verbessert werden.

1 Einleitung

Viele moderne Kryptowährungen ermöglichen anonymen Geldtransfer und benötigen keine zentralen Verwaltungsorgane. Sie schützen den Nutzer vor Regierungsgewalt und das Geld vor Wertverlust durch Inflation. Leider erfahren diese Währungen keine breite Akzeptanz. Kryptowährung ist ein recht junges Zahlungsmittel. Viele Menschen kennen diese Art von Währungen nicht. Zudem werden sie in den Medien meist mit kriminellen Machenschaften in Verbindung gebracht. Das größte Problem ist jedoch, dass es kaum Möglichkeiten gibt, mit diesen Währungen einzukaufen.
Diese Arbeit ist ein Proof of Concept (PoC). Sie befasst sich mit der Entwicklung eines Webshops, in dem mit der Kryptowährung Zcash[1] bezahlt werden kann. Der Webshop wird von Grund auf und ohne Content-Management-System (CMS) entwickelt. Sowohl das Front-End, als auch das Back-End, haben einen modularen Aufbau. Somit können Module beliebig ausgetauscht, kombiniert und ineinander integriert werden. Die Module des Front-Ends werden Webkomponenten (web component) genannt. Dagegen werden sie im Back-End, bzw. der Programmiersprache Go, Pakete (package) genannt. Das Front-End stellt den sichtbaren Webshop dar. Das Back-End beinhaltet Pakete, um Nutzer- und Walletdaten zu persistieren. Außerdem wird ein Paket für die Zahlungsabwicklung mit Zcash bereitgestellt. Ein Nutzer kann sich registrieren. Dabei werden seine Daten in einer Datenbank gespeichert. Nachdem der Nutzer sich eingeloggt hat, kann er einen Artikel im Webshop auswählen und ihn in den Warenkorb legen. Artikel im Warenkorb können später mit Zcash bezahlt werden. Die Differenz wird vom Wallet des Nutzers abgezogen und dem Verkäufer gutgeschrieben. Nach der Kaufabwicklung sieht der Nutzer, ob die Zahlung erfolgreich war.
Ausschließlich die Implementierung des Webshops und die Zahlungsabwicklung mit Zcash werden in dieser Arbeit beschreiben. Nicht beschrieben werden, das Deployment, Data-Mining, Währungstausch und das grafische Design. Das Responsive Webdesign des Front-Ends wird genau erläutert. Webshops sind die moderne Alternative zum Einzelhandel. Sie stellen eine einfache Variante dar, Zahlungen mit Kryptowährung zu ermöglichen.

1.1 Webkomponenten

Webkomponenten sind ein Überbegriff für browserbasierte APIs. Sie ermöglichen den Einsatz von Custom Elements, Shadow DOM, HTML Import und HTML Template. Diese Technologie ergänzt HTML, um wiederverwendbare, modulare und anpassbare Funktionen. Bisher werden Webkomponenten nur von Chrom und Opera vollständig unter-

[1] Zcash, https://z.cash/

stützt. Für Firefox und Edge stehen Polyfills[2] zur Verfügung.

1.2 Polymer 2

Polymer 2[3] ist die zweite Version des von Google Inc. entwickelten Front-End-Web-Frameworks. Das Ziel des Frameworks besteht darin, die Entwicklung von Webtechnologien voranzutreiben. Um dieses Ziel umzusetzen, wird der Umgang mit aktuellen, oder zukünftigen Technologien ermöglicht. Der Umgang mit Webkomponenten wird vereinfachen. Polymer bietet ES2015[4]-Unterstützung, auch ES6 genannt. ES2015 ist die aktuelle Version von JavaScript. Die Konfiguration von Service Workern wird durch eine Konfigurationsdatei ermöglicht. PolymerElements ermöglichen das Erstellen von gekapselten Custom Elements und unterstützen Datenbindung (data binding). Die Kommunikation zwischen Webkomponenten basiert auf Events. Diese Technologie ist nativ, durch den Browser möglich. Es werden keine weiteren Funktionen des Frameworks benötigt. Durch einen Transpiler wird ES2015-Code umgewandelt, sodass alle aktuellen Browser unterstützt werden. Polymer soll die stark JavaScript-basierten Frameworks ablösen. Um bessere Performance zu ermöglichen, wurde Polymer auf das Motto „use the plattform“ entwickelt. Das Nutzererlebnis verbessert sich ebenfalls, da der Benutzer nur auf das Laden der ersten Komponente (Shell) warten muss, um die Applikation zu bedienen.

1.3 Go

Go[5] ist eine kompilierende Programmiersprache, welche ebenfalls von Google Inc. entwickelt wurde. Von Rob Pike, Ken Thompson und Robert Griesemer stammen die Entwürfe. Ältere kompilierte Sprachen wie C und C++ ermöglichen performanten Code. Das Kompilieren verlangsamt den Entwicklungsprozess jedoch erheblich. Go wurde entwickelt, um dieses Problem zu lösen. Das Back-End dieser Arbeit, inklusive des eingebauten Webservers, kompiliert im Bruchteil einer Sekunde. In Go gibt es keine Klassen. Stattdessen gibt es Strukturen (struct), auf welche Methoden definiert werden können. Nebenläufigkeit wird durch Go-Routinen (goroutine) ermöglicht. Diese können durch Kanäle (channels) synchronisiert werden. Trotz seiner optischen Ähnlichkeit zu C ist Go eine grundlegend neue Sprache mit neuen Ansätzen. Mit Go wurde eine einfache, performante und produktive Sprache entwickelt.

[2] Polyfills [Erklärung], https://de.wikipedia.org/wiki/Polyfill
[3] Polymer 2, https://www.polymer-project.org/
[4] ES2015, https://developers.google.com/web/shows/ttt/series-2/es2015
[5] Go, https://golang.org/

1.4 Echo

Echo[6] ist ein von Vishal Rana entwickeltes Framework, welches die Webentwicklung mit Go vereinfacht. Echo enthält Methoden zur Verwaltung eines Webserver. Anfragen (Request) und Antworten (Response) des Webservers werden als Kontext (type struct echo.Context) übertragen. Dieser kann leicht in Strukturen konvertiert werden. Um Anfragen bzw. Kontext weiterzuleiten, bringt Echo einen passenden Router mit. Zudem enthält das Framework Middleware zur einfachen Handhabung von CORS, JWT, Logging und Recover-Funktionalität.

1.5 Zcash

Zcash ist eine Kryptowährung, welche private Transaktionen ermöglicht. Es basiert auf dem Bitcoin-Protokoll. Ergänzend besteht die Möglichkeit, anonyme Transaktionen durchzuführen. Die Anonymität wird durch das zk-SNARK[7]-Protokoll sichergestellt, welches auf Zero-Knowledge-Beweisen (zero-knowledge proof) basiert. Es wurde speziell für Zcash entwickelt. Zcash unterscheidet in transparente Adressen (t-address) und geschützte Adressen (z-address). Transparente Adressen ermöglichen die gleichen Transaktionen wie Bitcoin. Tatsächlich wird die gleiche API verwendet. Dabei werden Informationen in der Blockchain offengelegt und gespeichert. Geschützte Adressen ermöglichen dagegen Transaktionen, welche keine Informationen offenlegen. Da es sich um ein Zero-Knowledge-Protokoll handelt, ist kein Rückschluss auf die gesendete Menge, die Bilanz der Adresse, sowie die Adresse selbst möglich. Zero-Knowledge ist zum Zeitpunkt des Erstellens der Arbeit die stärkste Kryptografie. Es ist möglich Zcash zwischen transparenten und geschützten Adressen zu senden. Dabei werden ausschließlich Informationen der transparenten Adresse offengelegt.

[6] Echo, https://echo.labstack.com/
[7] zk-SNARK, https://z.cash/technology/zksnarks.html

2 Methoden

Ein Webshop besteht generell aus einem Front-End und einem Back-End. Das Front-End stellt die grafische Benutzeroberfläche dar. Es wurde Wert darauf gelegt, dem Nutzer ein gutes Erlebnis zu bereiten (user experience). Das Back-End beinhaltet Daten von Nutzern und Funktionalität des Webshops. Der Fokus liegt dabei auf Performance und Sicherheit. Die Zahlungsabwicklung mit Zcash wurde als ein Modul des Back-Ends implementiert.

2.1 Das Front-End

Ein Mensch geht öfter dahin und bleibt länger dort, wo er sich wohlfühlt. Genauso ist das im Web. Im Onlinehandel (E-Commerce) kann das zu höheren Umsätzen führen. Jeder hat schon mal eine schlechte Erfahrung mit einer Website gemacht. Dabei gibt es ein breites Spektrum an Ärgernissen, wie ein versteckter Button, ein komplizierter Bezahlprozess, oder eine schlechte Farbkombination. Solche Fehler verursachen Kosten für den Betreiber eines Webshop. Viele neue Technologien und Standards wurden entwickelt, um solchen Erlebnisse vorzubeugen und Webseiten zu optimieren. Einige wurde bei der Entwicklung dieses Webshops benutzt.
In den letzten Jahren überholte der Anteil der mobilen Endgeräte im Web, den der Desktop PCs. Die Webentwicklung wurde dementsprechend angepasst, sodass Responsive Webdesign entstand. Responsive (reagierend) Webdesign ermöglicht die Darstellung einer Webseite auf Endgeräten mit unterschiedlichen Bildschirmgrößen und Auflösungen. Dieser Webshop wurde nach dem Prinzip „mobile first“ entwickelt. D. h. der Webshop wurde entworfen, um gut auf kleinen Bildschirmen bedienbar zu sein. Zusätzlich basiert der Webshop auf Googles Material Design[8]. Darin werden Richtlinien für Abstände und Farbkombinationen definiert.
Web für mobile Endgeräte bringt weitere Herausforderungen mit sich. Das mobile Internet hat eine geringe Bandbreite, hohe Latenzzeiten, eine schlechte Erreichbarkeit und ist in den meisten Fällen volumenbegrenzt. Um diese Schwächen zu vermindern, wurde das Framework Polymer 2 verwendet. Der Einsatz von Shadow DOM ermöglicht es, die Applikation in kleinere Teile zu zerlegen. Da immer nur ein kleiner Teil der Webseite übertragen wird, sind die Ladezeiten gering. Diese Daten werden gespeichert und sind danach offline verfügbar. Die Größe einer mit Polymer entwickelten Webapplikation ist gering, da es Plattformfunktionen statt JavaScript nutzt. Das Benutzererlebnis verbessert sich erheblich, da der Nutzer schnell etwas von der Webseite sieht. Der zuerst geladene Teil der Applikation heißt Shell und sollte möglichst klein sein. So kann der Nutzer mit der Seite interagieren, auch wenn sie noch nicht vollständig geladen ist. Dabei stehen nicht alle Funktionen der Applikation zur Verfügung. Jedoch orientiert sich

[8] Material Design, https://material.io/

der Nutzer zunächst am bereits geladenen Kontext, sodass er nicht wartet, während die Seite lädt.
Die Implementierung basiert auf zwei neuen Webtechnologien. Das sind Shadow DOM und Service Worker. Sie werden zuerst erklärt.

2.1.1 Shadow DOM

Shadow DOM[9] ist einer von vier Standards der Webkomponenten. Es ermöglicht, das DOM (Data Object Model) in Teile zu gliedern. Webkomponenten können damit isoliert (HTML, CSS und JS) eingebunden werden. Durch den Einsatz von Shadow DOM ist CSS-Code nur für die Webkomponente sichtbar. Das CSS wird gekapselt. Damit wird die Entwicklung mit CSS einfacher und übersichtlicher. Um CSS-Regeln für eine Webkomponente, inklusive aller enthaltenen Komponenten festzulegen, wird die Pseudoklasse :host verwendet. Diese ist Teil der Spezifikation von Webkomponenten. CSS-Regeln werden global sichtbar, falls sie in der Shell unter :host aufgeführt werden. Das ist möglich, da alle anderen Webkomponenten aus der Shell aufgerufen werden. Es bietet sich an, CSS-Variablen in diese Pseudoklasse einzutragen.

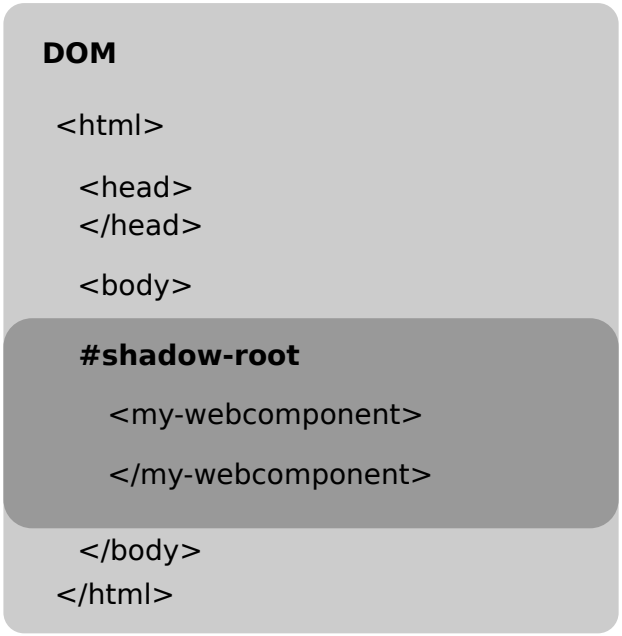

Abbildung 2.1: Shadow DOM

[9] Shadow DOM, https://www.w3.org/TR/shadow-dom/

2.1.2 Service Worker

Ein Service Worker[10] ist ein programmierbarer Proxy im Browser des Nutzers. Programmiert wird er typischerweise in JavaScript und läuft als Hintergrundprozess. Diese Webtechnologie ermöglicht es, offline Cache zu nutzen. Service Worker können auf verschiedene Arten implementiert werden, um bestimmte Anforderungen zu erfüllen. Eine typische Anwendung ist das Zwischenspeichern (caching) von Webseiten. Dabei wird zuerst der Service Worker gefragt, ob eine bestimme Ressource vorhanden ist. Ist das der Fall, wird sie aus dem Cache geladen. Falls nicht, wird eine Anfrage an den Webserver gesendet. Der Service Worker dieses Webshops funktioniert nach diesem Prinzip. Er ermöglicht die offline Nutzung des Webshops und verringert die übertragene Datenmenge.

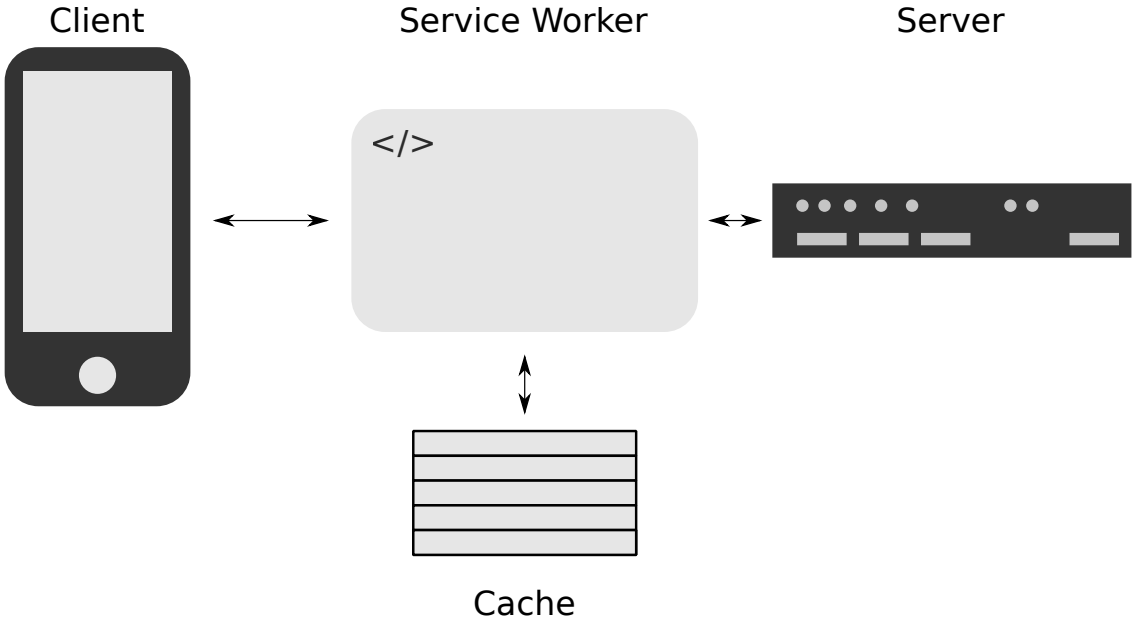

Abbildung 2.2: Service Worker

2.1.3 IndexedDB

IndexedDB[11] ist ein Standard, welche das Speichern von Daten im Browser ermöglicht. Vor IndexedDB wurden Daten in Cookies gespeichert. Manche Browser implementierten zudem eigene Speichermöglichkeiten. Bekannt sind LocalStorage und Sessionstorage. Diese Speicher wurden von verschiedenen Herstellern unterschiedlich implementiert und führte zu Problemen in der Webentwicklung. Eine standardisierte API machte Sinn und IndexedDB entstand. IndexedDB sollte asynchron implementiert werden, um

[10] Service Worker, https://developers.google.com/web/fundamentals/getting-started/primers/service-workers
[11] IndexedDB, https://www.w3.org/TR/IndexedDB/

die Webapplikation nicht zu blockieren. Für diesen Zweck wurden eigene Pakete entwickelt. In diesem Projekt wurde das Paket idb[12] (IndexedDB Promised) benutzt. Theoretisch ist das Paket nicht nötig. Es vereinfacht die Benutzung von IndexedDB jedoch sehr.

2.1.4 Aufbau

Das Front-End ist nach Polymers Standardaufbau konzipiert. Für diesen Standard gibt es eine Vorlage, welche über das Polymer-CLI (Polymer Command Line Interface) erstellt wird. Durch den folgenden Befehl werden alle Komponenten heruntergeladen und Konfigurationsdateien erstellt. Kommandozeilenbefehle werden in dieser Arbeit durch ein „$“-Zeichen gekennzeichnet.

```
$ polymer init polymer-2-application
```

Diese Vorlage beinhaltet die Dateien und Verzeichnisse bower_components/, bower.json, index.html, polymer.json, README.md, src/ und test/. Dateien werden in dieser Arbeit mit der Dateiendung angegeben. Verzeichnisse werden durch den Suffix „/“ gekennzeichnet.
Abbildung 2.3 zeigt die oberste Ebene der Architektur des Front-Ends. Die Dateien app.yaml, Dockerfile und manifest.json sind Teil des Deployments und werden nicht näher erläutert. Das Verzeichnis test/ beinhaltet die Tests des Front-Ends. Es wird ebenfalls nicht erläutert. Unter images/ werden die Bilder, der im Webshop angebotenen Artikel gespeichert. Diese dienen nur der Präsentation, da Artikeldaten in Zukunft von einem Drittanbieter geladen werden. Alle anderen Dateien und Verzeichnisse werden nun

```
app.yaml
bower_components/
bower.json
build/
Dockerfile
images/
index.html
manifest.json
node_modules
polymer.json
README.md
service-worker.js
src/
sw-precache-config.js
test/
```

Abbildung 2.3: Aufbau des Front-Ends

genauer erklärt. Bower[13] ist ein Paketverwaltungstool für Webentwickler. Polymer be-

[12] idb — IndexedDB Promised, https://www.npmjs.com/package/idb
[13] Bower — a package manager for the web, https://bower.io/

nutzt dieses Tool als Paketverwaltung. Alle Komponenten von Polymer und alle Bower-Pakete werden in das Verzeichnis bower_components geladen. Aus diesem Verzeichnis werden später die Webkomponenten geladen. In Abbildung 2.4 wird der Inhalt der Datei bower.json gezeigt. Das ist die Konfigurationsdatei von Bower. Name, Beschrei-

```
{
  "name": "store",
  "description": "Store Application",
  "main": "index.html",
  "dependencies": {
    "polymer": "Polymer/polymer#^2.0.0-rc.7",
    "webcomponentsjs": "webcomponents/webcomponentsjs#^1.0.0",
    "app-route": "PolymerElements/app-route#2.0.0",
    "app-layout": "PolymerElements/app-layout#2.0.0",
    "app-storage": "PolymerElements/app-storage#2.0.0",
    "iron-icon": "PolymerElements/iron-icon#2.0.0",
    "iron-iconset-svg": "PolymerElements/iron-iconset-svg#2.0.0",
    "iron-pages": "PolymerElements/iron-pages#2.0.0",
    "iron-ajax": "PolymerElements/iron-ajax#2.0.0",
    "iron-flex-layout": "PolymerElements/iron-flex-layout#2.0.0",
    "iron-image": "PolymerElements/iron-image#2.0.0",
    "paper-button": "PolymerElements/paper-button#2.0.0",
    "paper-icon-button": "PolymerElements/paper-icon-button#2.0.0",
    "paper-input": "PolymerElements/paper-input#2.0.0",
    "paper-item": "PolymerElements/paper-item#2.0.0"
  },
  "devDependencies": {
    "web-component-tester": "v6.0.0-prerelease.5"
  }
}
```

Abbildung 2.4: frontend/bower.json

bung und die Startdatei werden zuerst angegeben. Danach folgen die Abhängigkeiten. Abhängigkeiten werde in Software-Abhängigkeiten (dependencies) und Entwickler-Abhängigkeiten (devDependencies) unterteilt. Die Entwickler-Abhängigkeiten werden beim Build-Prozess nicht berücksichtigt. Benötigte Pakete können in diese Datei eingetragen und installiert werden. Um nicht erst eine alte Version des Pakets herunterzuladen, sollten die bereits installierten Pakete zuerst aktualisieren werden. Bower lädt automatisch die neuste Version des Pakets, welche alle Abhängigkeiten erfüllt.

```
$ bower update
$ bower install
```

Alternativ können Pakte per Befehl installiert werden. In diesem Beispiel wird die Webkomponente paper-input aus der Sammlung (collection) PolymerElements installiert.

```
$ bower install --save PolymerElements/paper-input
```

Das Element wird heruntergeladen, in das Verzeichnis bower_components/ verschoben und ein Eintrag in bower.json angelegt. Die Option –save gibt an, dass das Element

eine Software-Abhängigkeit ist und in bower.json eingetragen wird.
Das Verzeichnis node_modules/ hat eine ähnliche Funktion wie bower_components/. Es gehört zu einer zweiten Paketverwaltung mit dem Namen npm[14] (Node Package Manager). Das Entwicklerteam von Polymer hat sich für Bower entschieden, um ihre Webkomponenten zu verwalten. Npm ist jedoch das größte und bekannteste Paketverwaltungstool für die Entwicklung von Webapplikationen. Tatsächlich werden Bower und Polymer-CLI zuerst durch npm installiert. Die Option -g bedeutet, dass das Paket global bzw. systemweit installiert wird. Unter Linux muss dieser Befehl mit sudo-Rechten ausgeführt werden.

```
$ npm install -g bower
$ npm install -g polymer-cli
```

Die Pakete der beiden Paketverwaltungstools können beliebig kombiniert werden. Sie werden unabhängig von einander verwaltet und ins Projekt importiert. Bei der plattformnahen Entwicklung mit Polymer werden JavaScript-basierte Frameworks selten verwendet.
Sw-precached-config.js und service-worker.js bilden den Service Worker.
Die Datei polymer.json ist die Konfigurationsdatei für Polymer. Alle wichtigen Informationen, die das Framework für den Build-Prozess benötigt, stehen in dieser Datei. Abbildung 2.5 zeigt den Inhalt der Datei. Zuerst wird darin die Startdatei (hier entrypoint) der Webapplikation angegeben. Danach folgt ein Eintrag für die Shell. Diese Datei wird direkt nach der index.html geladen.
Die Fragmente (fragments) stellen, genau wie die Shell, eine Webkomponente dar. Die Komponenten des Webshops wurden in store-app und store-pages unterteilt. Store-app enthält die Shell und Webkomponenten für die Navigation. Store-pages stellen den Content der Seite dar. Auffällig ist, dass alle Webkomponenten mit dem Präfix „store-“ anfangen. Diese Art der Namensgebung ist in der Spezifikation von Webkomponenten festgelegt. Dadurch werden doppelte Namen vermieden. Webkomponenten werden als HTML-Element eingebunden. Dazu dient lediglich der Name der Komponente. Ein unterschiedlicher Pfad zur Datei hat damit keinen Einfluss auf den Namen. Das Minimalbeispiel (Abb. 2.6) zeigt, wie die Webkomponente store-shell in die Datei index.html eingebunden wird. Eine Webkomponente ist ein HTML-Element (custom element), welches vom Entwickler definiert wird. Diese Elemente funktionieren durch APIs des Browsers. Es wird kein externes Framework integriert. In dieser Arbeit werden HTML-Elemente der Form <element></element> als <element> geschrieben.

```
<html>
  - Content -    =>     <html>
</html>
```

Webkomponenten ermöglichen wiederverwendbaren HTML-Code. Es wird weniger JavaScript als bei herkömmlichen SPAs (single-page application) benötigt. Daraus folgt

[14] npm — the package manager for JavaScript, https://www.npmjs.com/

```
{
  "entrypoint": "index.html",
  "shell": "src/store-app/store-shell.html",
  "fragments": [
    "src/store-app/store-header.html",
    "src/store-app/store-left-drawer.html",
    "src/store-app/store-right-drawer.html",
    "src/store-pages/store-login.html",
    "src/store-pages/store-register.html",
    "src/store-pages/store-logout.html",
    "src/store-pages/store-account.html",
    "src/store-pages/store-basket.html",
    "src/store-pages/store-orders.html",
    "src/store-pages/store-wallet.html",
    "src/store-pages/store-smartphones.html",
    "src/store-pages/store-notebooks.html",
    "src/store-pages/store-not-found.html"
  ],
  "sources": [
    "src/**/*",
    "images/**/*",
    "bower.json"
  ],
  "extraDependencies": [
    "manifest.json",
    "bower_components/webcomponentsjs/*.js"
  ],
  "lint": {
    "rules": ["polymer-2"]
  },
  "builds": [{
    "preset": "es5-bundled"
  }]
}
```

Abbildung 2.5: frontend/polymer.json

eine bessere Performance.
Unter sources werden alle Verzeichnisse angegeben, die Quellcode oder Ressourcen beinhalten. Der einfache Stern steht dabei für eine beliebige Datei. Der doppelte Stern gibt an, dass sich die Dateien in beliebig vielen Unterverzeichnissen befinden können. Alle Dateien aus src/, images/ und deren Unterverzeichnissen werden geladen.
ExtraDependencies sind Abhängigkeiten, von denen Polymer abhängt.
Ein Linter ist ein Tool der statischen Codeanalyse. Damit können Schreib- und Syntaxfehler während der Entwicklung angezeigt werden. In vielen Entwicklungsumgebungen werden Fehler rot dargestellt. Diese Fehler führen dazu, dass die Webapplikation nicht funktioniert. Der Interpreter gibt einen Error zurück. Warnungen werden meist gelb, oder orange dargestellt. Sie führen nicht zum Abbruch des Interpreters, jedoch zu unsauberem Code.
Am Ende der Datei ist der Eintrag builds aufgeführt. Hier werden alle Optionen des Build-Prozesses aufgelistet. Polymer bietet vordefinierte Sammlungen (preset) an. Die-

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <script src="bower_components/webcomponentsjs/webcomponents-loader.js">
    </script>
    <link rel="import" href="src/store-app/store-shell.html">
  </head>
  <body>
    <store-shell></store-shell>
  </body>
</html>
```

Abbildung 2.6: Minimalbeispiel: Webkomponenten einbinden

se stellen vernünftige Kombinationen der Build-Optionen dar. Abbildung 2.7 zeigt die Optionen des Presets es5-bundled. Diese Optionen wurden gewählt, um eine gute

```
name: es5-bundled
js: {minify: true, compile: true}
css: {minify: true}
html: {minify: true}
bundle: true
addServiceWorker: true
addPushManifest: true
insertPrefetchLinks: true
```

Abbildung 2.7: polymer.json preset es5-bundled

Browserunterstützung zu erzielen.
HTML-, CSS- und JS-Dateien werden vom Server unverändert übertragen. Diese Dateien werden nicht kompiliert. Wenn Funktions- und Variablennamen gekürzt werden, verringert sich die übertragene Datenmenge. Dazu dienen Minifier. Der Code wird kleiner und performanter. Der umgewandelte Code ist für Menschen nicht mehr lesbar.
ES2015 (ECMAScript 2015) ist die aktuelle Version von JavaScript. Sie wird noch nicht von allen Browsern unterstützt. Da es die Entwicklung mit JavaScript erleichtert, schreibt man ES2015 Code und kompiliert ihn zu ES5. ES5 wird von allen aktuellen Browsern unterstützt. Polymer kompiliert ES2015 zu ES5, wenn „compile: true“ gesetzt wurde.
Um die Anzahl der HTTP-Anfragen zu verringern, werden Dateien zusammengefasst (bundle). Andernfalls fragt der Client nach der HTML-Datei und erst werden sequenziell CSS- und JS-Dateien geladen. Jeder Datei wird einzeln und nacheinander angefragt. Das verringert die Performance der Webapplikation. HTTP/2 Server Push wurde entwickelt, um dieses Problem zu beheben. Der Client kann durch diese Funktionalität asynchron Dateien beim Server anfragen. Bis HTTP/2 eine größere Akzeptanz erreicht, ist Bundling die beste Alternative.
AddServiceWorker erstellt den Service Worker. Dieser wird später genauer erläutert.
AddPushManifest generiert eine Auflistung von Dateien für HTTP/2 Server Push. Der Server unterstützt Push jedoch nicht. Diese Option erfüllt im Moment keinen Zweck.

Das PushManifest schadet allerdings auch nicht.
InsertPrefetchLinks teilt Polymer mit, dass Fragmente im Voraus geladen werden (lazy load). Somit werden die Wartezeiten für den Nutzer verringert. Das Nutzererlebnis verbessert sich.

2.1.5 Implementierung

In diesem Kapitel wird die Implementierung des Front-Ends anhand des Quellcodes beschrieben. Darin werden Aussehen und Funktionalität des Webshops festgelegt. Der Code wird ausführlich erklärt. In Abbildung 2.8 sind alle Dateien aufgeführt, welche Quellcode enthalten. Eingerückte Dateien befinden sich im darüber stehendem Verzeichnis. Zuerst fällt auf, dass die Implementierung ausschließlich HTML-Dateien ent-

```
index.html
src/
store-app/
  store-header.html
  store-icons.html
  store-left-drawer.html
  store-right-drawer.html
  store-shell.html
store-pages/
  form-styles.html
  store-account.html
  store-basket.html
  store-login.html
  store-logout.html
  store-notebooks.html
  store-not-found.html
  store-orders.html
  store-register.html
  store-smartphones.html
  store-wallet.html
```

Abbildung 2.8: Quellcodedateien

hält. Der Grund dafür ist das Framework Polymer 2. In Polymer stellt jede Quellcodedatei eine Webkomponente dar. Webkomponenten können beliebig ineinander verschachtelt werden. Polymer legt den Aufbau einer Webkomponente fest. Diese Webkomponenten werden Polymer-Elemente genannt. Abbildung 2.9 zeigt dem Aufbau eines Polymer-Elements. Alle in Abbildung 2.8 aufgeführten Dateien haben diesen Aufbau. Eine Ausnahme ist die Datei index.html.
Ein Polymer-Element beginnt mit dem Import anderer Webkomponenten und ausgelagerter CSS-Dateien. Der Import erfolgt als <link> mit dem Attribut rel=”import”.

```
<link rel="import" href="../bower_components/polymer/polymer-element.html">
```

Alle Webkomponenten, die das Polymer-Element benutzt, sollten aufgeführt werden.

```
<link rel="import" href="../bower_components/polymer/polymer-element.html">

<dom-module id="app-element">
<template>
<style>
  :host {
    display: block;
  }
</style>

<div>
  <p>Some HTML</p>
</div>
</template>

<script>
class AppElement extends Polymer.Element {
  static get is() { return 'app-element'; }

  static get properties() {
    return {};
  }
}
window.customElements.define(AppElement.is, AppElement);
</script>
</dom-module>
```

Abbildung 2.9: Polymer Element

Wurde eine Webkomponente von einem vorher geladenen Element importiert, ist das theoretisch nicht nötig. Allerdings führen nicht importierte Elemente schnell zu unerwarteten Fehlern. Durch das Weglassen von Imports kann keine Performancesteigerung erzielt werden, da zuvor geladene Webkomponenten durch den Service Worker importiert werden. Global benutzte JavaScript-Bibliotheken und -Frameworks werden in der Datei index.html importiert.
Danach folgt das <dom-module>.

```
<dom-module id="app-element">
```

Das <dom-module> umschließt das Polymer-Element wie ein Container. Dieser Container kann später in das Shadow DOM eingebunden werden. Durch das Attribut id wird der Name des Elements festgelegt. Er muss mit dem zurückgegebenen Namen der zugehörigen JS-Klasse übereinstimmen.
Das Template beinhaltet <style>, gefolgt vom HTML-Code der Webkomponente.

```
<template>
  <style></style>
  <p>Some HTML</p>
</template>
```

In <style> wird der CSS-Code der Webkomponente eingefügt.

```
<style>
  :host {
    --primary-light-color: #fff;
  }
</style>
```

Darunter kann beliebiger HTML-Code eingefügt werden.
Als nächstes folgt <script>. Darin wird der JS-Code implementiert. Da Polymer ES2015 unterstützt, ist das i.d.R. eine Klasse. Diese Klasse erbt von Polymer.Element.

```
class AppElement extends Polymer.Element {}
```

In der ersten Zeile der Klasse wird eine get-Methode definiert. Sie gibt den Namen der Webkomponente zurück. Die Methoden get und set sind Teil der ES2015 Spezifikation.

```
static get is() { return 'app-element'; }
```

Danach folgt ein Getter für die Eigenschaften (property) der Klasse.

```
static get properties() { return {}; }
```

Zuletzt wird das Polymer-Element als Webkomponente definiert. Dadurch werden die JavaScript-Klasse und das Template verbunden.

```
window.customElements.define(AppElement.is, AppElement);
```

Die grundlegende Implementierung eines Polymer-Elements bzw. einer Webkomponente des Polymer Frameworks wurde nun erklärt. Als nächstes werden die wichtigsten Quellcodedateien im Detail erläutert. Es ist nicht möglich, den gesamten Quellcode dieser Arbeit zu erklären. Der vorgestellte Code enthält die wichtigsten Teile der Implementierung. Dabei stehen die Entwicklung mit Polymer und die Kernfunktionen des Webshops im Vordergrund.
Die Datei index.html ist die Startdatei der Webapplikation.

Listing 2.1: index.html

```
<!doctype html>
<html lang="en">
  <head>
  <meta charset="utf-8">
  <meta name="generator" content="CrypStore">
  <meta name="viewport" content="width=device-width, minimum-scale=1,
      initial-scale=1, user-scalable=yes">
  <title>CrypStore</title>
  <meta name="description" content="Zcash Webshop">
```

```
  <base href="/">
  <link rel="icon" href="images/favicon.ico">
  <link rel="manifest" href="manifest.json">
  <script src="node_modules/idb/lib/idb.js"></script>
  <script src="bower_components/webcomponentsjs/webcomponents-loader.js"></
      script>
  <link rel="import" href="src/store-app/store-shell.html">

  <!--service worker-->
  <script>
    const baseUrl = document.querySelector('base').href;
    if ('serviceWorker' in navigator) {
      indow.addEventListener('load', function() {
        navigator.serviceWorker.register(baseUrl + 'service-worker.js');
      });
    }
  </script>

  <!-- Add any global styles for body, document, etc. -->
  <style>
    body {
      margin: 0;
    font-family: 'Roboto', 'Noto', sans-serif;
  }
  </style>
  </head>
  <body>
  <store-shell></store-shell>
  <noscript>
    Please enable JavaScript to view this website.
  </noscript>
  </body>
</html>
```

Am Anfang der Datei wird der Dokumententyp deklariert. Das ist eine Konvention für HTML-Dateien, jedoch nicht notwendig. Danach folgt <html>, die Wurzel der HTML-Datei. Es beinhaltet einen <head> und <body>. Im <head> stehen Metainformationen wie Textkodierung, Name und Beschreibung. Außerdem werden JavaScript-Bibliotheken und die Shell importiert.

```
<script src="node_modules/idb/lib/idb.js"></script>
<script src="bower_components/webcomponentsjs/webcomponents-loader.js"></
    script>
<link rel="import" href="src/store-app/store-shell.html">
```

Darauf folgt die Registrierung des Service Workers und ein style-Element, worin alle globalen CSS-Regeln aufgeführt werden. In diesem Fall wird margin: 0 gesetzt, da manche Browser Standardwerte ungleich 0 verwenden. Darunter folgt die Schriftart des Webshops. Der <body> enthält die Shell <store-shell> und eine Nachricht, die ange-

zeigt wird, falls der Nutzer JavaScript deaktiviert hat.
Als nächstes werden die wichtigsten Webkomponenten des Shops erläutert, beginnend mit der Shell <store-shell>. Der Aufbau eines Polymer-Elements wurde am Anfang des Kapitel beschrieben. HTML und JavaScript werden unabhängig voneinander erklärt. Dabei wird mit HTML begonnen. Der Code wird nicht vollständig aufgeführt. Imports werden weggelassen und HTML-Content auf das Minimum reduziert. JS-Funktionen werden soweit gekürzt, dass sie die Bedeutung des Codes wiedergeben. Sie müssen nicht vollständig, oder syntaktisch korrekt sein.
Die Shell trägt den Namen <store-shell> und ist die einzige Datei, welche in die Startdatei index.html eingebunden ist. In ihr befindet sich das Layout der App, das Routing und Events werden von hier aus zu Unterkomponenten verschickt. Die Elemente <app-location>, <app-route>, <app-drawer>, <app-drawer-layout>, <app-header>, <app-header-layout> und <iron-pages> wurden vom Polymer Team entwickelt. Deren Quellcode ist offen und frei verfügbar. Das ist der größte Vorteil von Webkomponenten. Sie können beliebig ausgetauscht und von Entwicklern weltweit veröffentlicht werden. Webcomponents.org ist die größte Verwaltungsseite von Webkomponenten.

Listing 2.2: <store-shell>

```
<app-location route="{{route}}"></app-location>
<app-route
  route="{{route}}"
  pattern="[[rootPattern]]:page"
  data="{{routeData}}"></app-route>
<app-drawer-layout fullbleed on-narrow-changed="_narrowChanged">
  <app-drawer id="drawer" align="right" slot="drawer" swipe-open>
    <store-right-drawer page={{page}}></store-right-drawer>
  </app-drawer>
  <app-header-layout has-scrolling-region>
    <store-header id="header" hidden="[[!narrow]]"></store-header>
    <!--main content-->
    <iron-pages
      selected="[[page]]"
      attr-for-selected="name"
      fallback-selection="not-found"
      role="main"
      on-selected-item-changed="_rerender">
      <store-login name="login"></store-login>
    </iron-pages>
    <store-left-drawer page={{page}} toggle="[[leftToggle]]"></store-left-
        drawer>
  </app-header-layout>
</app-drawer-layout>
```

In <app-location> wird die Route (route) angegeben. Das ist der Teil der URL, der nach der Domain folgt. Es steht in direkter Verbindung zu <app-route>. Die Route wird in mehrere Abschnitte unterteilt. Das Pattern beschreibt, wie die Route unterteilt ist. Hier folgt die Seite direkt nach der Wurzel des Pfades „/“. Im Moment dient nur die Seite (page)

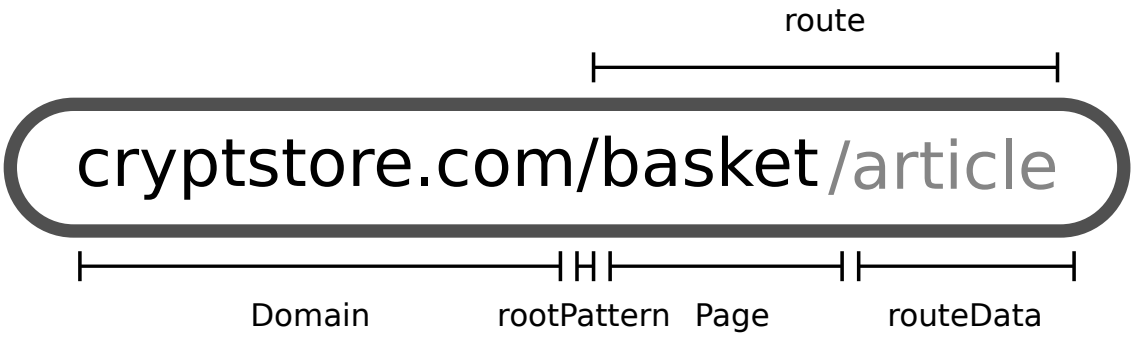

Abbildung 2.10: <app-route>

zur Navigation im Webshop. Es können jedoch mehr Abschnitte hinzugefügt werden. In der Abbildung 2.10 zeigt das leicht ausgegraute „/article“ eine mögliche Erweiterung.
Das Layout des Webshops besteht aus zwei <app-drawer> und einem <app-header>. Abbildung 2.11 zeigt, wie diese Webkomponenten angeordnet sind. Sie werden von <app-drawer-layout> umschlossen. Das Drawer-Layout passt das Layout der Seite an den durch slot=”drawer” gekennzeichneten Drawer an. Beispielsweise wird der Inhalt der Seite eingerückt, wenn der Drawer ein- und ausgeblendet wird. Der Header ist das obere Feld mit der Suchleiste. Die Drawer sind jeweils links und rechts.
Nun wird die Klasse der Shell beschrieben. Sie enthält die Eigenschaften page, rootPat-

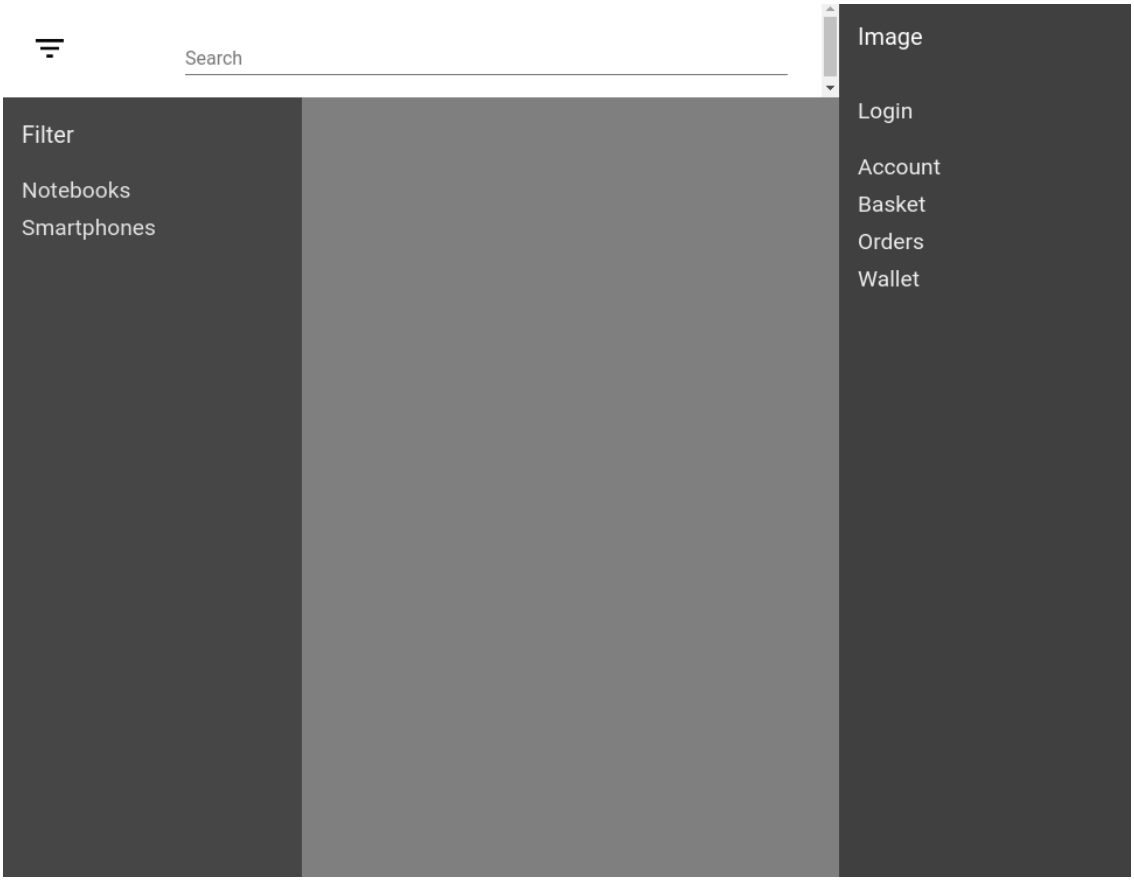

Abbildung 2.11: Layout des Webshops

tern und routeData. Diese Eigenschaften sind an die HTML-Attribute von <app-route> gebunden. Page ist vom Typ String, also eine Zeichenkette. Wenn sich das Attribut ändert, ändert sich die Eigenschaft (data binding). Das geschieht auch, wenn kein Event verschickt wurde (reflecttoattribute: true). Die Beobachtermethode (observer) führt die angegebene Methode _pagechanged immer aus, wenn sich die Eigenschaft ändert. Diese Methode lädt die angefragte Seite. Falls die angefragte Seite nicht existiert, wird die Seite <store-not-found> aufgerufen. Sie enthält die Nachricht „Page not found. 404“. Falls nur der Typ einer Eigenschaft angegeben wird, genügt die folgende Schreibweise.

```
property: type,
```

Die Methode ready() ist Teil des Lebenszyklus (lifecycle) eines Polymer-Elements. Sie wird durch Polymer definiert. Es gibt drei weitere Lebenszyklusmethoden: constructor(), connectedCallback() und disconnectedCallback(). Im Gegensatz zu ready() gehören sie zur Spezifikation von Webkomponenten. Die geerbte Funktion muss zuerst aufrufen werden (super.Funktionsname()), um sie dann zu überschreiben. Wurde die Komponente vollständig geladen, wird ready() ausgeführt. Darin wird ein EventListener hinzugefügt. Er lauscht auf das Event 'toogle-right'. Wird dieses Event verschickt, ruft der EventListener die Methode this.$.drawer.toggle() auf. Sie öffnet und schließt den Drawer.
Die Methode dispatchEvent(Event) sendet ein neuen Event. Events können an alle, oder eine ausgewählte Webkomponente verschickt werden. _rerender(e) sendet das Event 'render-page' an <store-basket>.

```
const page = this.shadowRoot.querySelector('store-basket');
page.dispatchEvent(new CustomEvent('render-page'));
```

Methoden mit dem Präfix „_“ sind als privat gekennzeichnet. JavaScript unterstützt keine tatsächlich privaten Methoden. Es ist jedoch eine Konvention. Die Methode _loggedIn() meldet, dass sich der Nutzer erfolgreich eingeloggt hat und ruft <store-basket> auf.

Listing 2.3: StoreShell

```
class StoreShell extends Polymer.Element {
  static get is() { return 'store-shell'; }
  static get properties() {
    return {
      page: {
        type: string,
        reflecttoattribute: true,
        observer: '_pagechanged',
      },
      rootPattern: string,
      routeData: object,
    };
```

```
  }
  ready() {
    super.ready();
    this.$.header.addeventlistener('toggle-right', e => this.$.drawer.toggle
        ());
  }
  _pagechanged(page) {
    const resolvedpageurl = this.resolveurl('../store-pages/store-' + page +
         '.html');
      polymer.importhref(
        resolvedPageUrl,
        null,
        this._showPage404.bind(this),
        true);
  }
  _rerender(e) {
    const page = this.shadowRoot.querySelector('store-basket');
    page.dispatchEvent(new CustomEvent('render-page'));
  }
  _loggedIn() {
    const page = this.shadowRoot.querySelector('store-right-drawer');
    page.dispatchEvent(new CustomEvent('loggedIn'));
    this.page = 'basket';
  }
}
```

Der Header des Webshops wurde <store-header> genannt und als Toolbar <app-toolbar> implementiert. Er beinhaltet zwei Buttons und eine Eingabezeile. Die Buttons erscheinen als Icons und implementieren ein on-click-Event. Diese Events rufen eine Methode auf, wenn der Button gedrückt wird. Die Eingabezeile dient als Suchleiste. Es wurde bisher keine Suchfunktion implementiert.

Listing 2.4: <store-header>

```
<app-header slot="header" shadow fixed>
  <app-toolbar>
  <paper-icon-button icon="store-icons:filter" on-click="toggleLeft">
  </paper-icon-button>
  <paper-input label="Search" type="search"></paper-input>
  <paper-icon-button icon="store-icons:menu" hidden="[[hidden]]"
    on-click="toggleRight"></paper-icon-button>
  </app-toolbar>
</app-header>
```

Die Klasse StoreHeader beinhaltet die Eigenschaft hidden. Sie macht den Button <paper-icon-button> unsichtbar, wenn der Drawer offen ist. Der Button öffnet und schließt den rechten Drawer. Auf Smartphones wird der Drawer geschlossen, wenn der Nutzer den Seiteninhalt antippt, oder mit dem Finger nach rechts zeiht. Auf größeren Monitoren ist der rechte Drawer dauerhaft geöffnet. Die Methode toggleRight() sendet ein spezielles

Event. Events werden gewöhnlich nur an die Webkomponete selbst und Unterkomponenten geschickt. Durch die Optionen „{bubbles: true, composed: true}“ wird das Event über die Shadow Root hinaus versendet.

Listing 2.5: StoreHeader

```
class StoreHeader extends Polymer.Element {
  static get is() { return 'store-header'; }
  static get properties() {
    return {
      hidden: Boolean,
    };
  }
  toggleRight() {
    this.dispatchEvent(
    new CustomEvent('toggle-right', {bubbles: true, composed: true}));
  }
}
```

Der rechte Drawer wurde als <app-drawer> in die Shell implementiert, um die Layoutfunktionalität zu nutzen. Die Webkomponente <store-right-drawer> stellt den Inhalt des Drawers bereit. Zuerst wurde eine Toolbar eingefügt, welche später das Logo des Shops zeigen soll. Danach folgen Vorlagen/Templates. Sie sind Teil des Polymer-Frameworks. Die folgenden zwei Templates stehen zur Verfügung.

```
<template is="dom-if" if="[[Condition]]"></template>
<template is="dom-repeat" items="[[Array]]"></template>
```

Das Erste ist ein bedingtes Template. Es wird nur gezeigt, falls die Bedingung erfüllt wird. Das zweite Template ist eine Art Schleife. Es iteriert über ein Array und gibt dessen Inhalt aus. Durch die bedingten Templates zeigt der Drawer „Login“, falls der Nutzer nicht eingeloggt ist. Andernfalls zeigt er „Logout“. Das letzte Template iteriert über das Array userContent und fügt die Einträge zum Drawer hinzu.

Listing 2.6: <store-right-drawer>

```
<app-toolbar>Image</app-toolbar>
<iron-selector selected="[[page]]" attr-for-selected="name" role="navigation
    ">
 <template is="dom-if" if="[[!loggedIn]]">
   <app-toolbar><a href="/login" name="login">login</a></app-toolbar>
 </template>
 <template is="dom-if" if="[[loggedIn]]">
   <app-toolbar><a href="/logout" name="logout" on-tap="logout">logout</a></
       app-toolbar>
 </template>
 <template is="dom-repeat" items="[[userContent]]">
   <a href="/[[item]]" name="[[item]]">[[item]]</a>
 </template>
</iron-selector>
```

Neben userContent und loggedIn implementiert StoreRightDrawer die Eigenschaft page. Sie wird an die Shell übertragen, falls eine Seite des Drawers ausgewählt wurde. Somit wird der ausgewählte Content geladen. StoreRightDrawer implementiert die Lebenszyklusmethode conntectedCallback(). Darin wird die IndexedDB tokenDB geöffnet. Falls sich in dessen ObjectStore tokenStore ein Token befindet, ist der Nutzer eingeloggt. Die Eigenschaft loggedIn wird auf true gesetzt. Die Methode ready() fügt einen EventListener hinzu, welche auf das Event loggedIn lauscht. Sie setzt die Eigenschaft loggedIn ebenfalls auf true.

Listing 2.7: StoreRightDrawer

```
class StoreRightDrawer extends Polymer.Element {
  static get is() { return 'store-right-drawer'; }

  static get properties() {
    return {
      page: {
        type: String,
        reflectToAttribute: true,
      },
      userContent: {
        type: Array,
        value() {
          return [
            'account',
            'basket',
            'orders',
            'wallet',
          ];
        },
      },
      loggedIn: {
        type: Boolean,
        value: false,
      },
    };
  }

  connectedCallback() {
    super.connectedCallback();
    // if token in indexedDB, user is logged in
    idb.open('tokenDB', 1, upgradeDB => {
      if (!upgradeDB.objectStoreNames.contains('tokenStore')) {
        upgradeDB.createObjectStore('tokenStore', {autoIncrement: true});
      }
    }).then(db => {
      const tx = db.transaction('tokenStore', 'readonly');
      const count = tx.objectStore('tokenStore').count();
      return count;
    }).then(count => {
      if (count != undefined && count > 0) {
        this.loggedIn = true;
```

```
      }
    });
  }

  ready() {
    super.ready();
    this.addEventListener('loggedIn', e => {this.loggedIn = true});
  }
}
```

Die Webkomponente <store-basket> stellt den Warenkorb (basket) des Webshops dar. Alle Content-Seiten werden von einem umschlossen, welches class=”container” enthält. Darin wird der Abstand von Content zum Rest der Seite definiert. Danach folgt ein bedingtes Template, welches angezeigt wird, falls der Nutzer erfolgreich eine Bestellung durchgeführt hat. Das nächste Template iteriert über basket. Diese Eigenschaft ist ein Array, welches alle vom Nutzer hinzugefügten Artikel enthält. Es zeigt ein Bild, den Namen und den Preis des Artikels. Im Abschnitt Design (Abb. 2.12) ist der Content dieser Webkomponente, der Warenkorb, zu sehen. Am Ende der Liste steht ein <paper-item>. Darin wird die Summe des Einkaufs angegeben. Zuletzt folgt ein Submit-Button. Damit bestätigt der Nutzer seinen Einkauf. Die Applikation schickt eine Anfrage an das Back-End.

Listing 2.8: <store-basket>

```
<div class="container">
  <template is="dom-if" if="[[orderComplete]]">
    <p>Order complete.</p>
  </template>
  <template is="dom-repeat" items={{basket}}>
    <paper-item>
      <iron-image sizing="contain" preload fade src="[[item.value.article.
          image]]"></iron-image>
      <div class="right">
        <p>[[item.value.name]]</p>
        <p>[[item.value.article.price]] ZEC</p>
      </div>
      <paper-input label="qty" value="{{item.value.article.qty}}"></paper-
          input>
    </paper-item>
  </template>
  <paper-item id="card-value">
    <iron-icon icon="store-icons:shopping-cart"></iron-icon>
    <div> Shopping Cart Value </div>
    <paper-input id="sum" label="ZEC" value="[[sum]]" readonly></paper-input
        >
  </paper-item>
  <paper-button id="submit" raised  type="submit" on-click="onBuy">buy</
      paper-button>
</div>
```

Die Klasse StoreBasket enthält das Array basket, worin Artikel gespeichert werden. Außerdem beinhaltet es die Zahl sum. Darin wird der Gesamtpreis des Einkaufs gespeichert. Da es sich um Zcash handelt, hat diese Zahl 6 Nachkommastellen. Desweiteren gibt es die Eigenschaft orderComplete. Sie wird auf true gesetzt, wenn der Einkauf erfolgreich war. Basket implementiert einen speziellen Observer. Er überwacht alle Änderungen innerhalb des Arrays. Da ein ganzer Pfad überwacht wird, muss ein get definiert werden, der alle Observer zurückgibt. Der „*“ kennzeichnet, dass alle Inhalte des Array überwacht werden.

```
static get observers() {
  return [
    'basketChanged(basket.*)',
  ];
}
```

Falls sich der Inhalt ändert, wird die Methode basketChanged(item) aufgerufen. Der geänderte Artikel wird aus dem Array basket geholt und in die IndexedDB articleDB geschrieben. Danach wird der Gesamtpreis neu berechent. ConnectedCallback() führt die Methode _renderPage() aus. Das ist notwendig, um den Inhalt des Warenkorbs korrekt anzuzeigen, wenn er zum ersten Mal aufgerufen wird. Die Mehtode ready() fügt einen EventListener für _renderPage() hinzu. _renderPage() führt zuerst die Methode _getArticles() aus. Sie holt alle Artikel aus der articleDB mit dem API-Aufruf getAll(). Wenn _getArticles() die Daten geholt hat, gibt sie ein JavaScript-Promise zurück. So wird sichergestellt, dass alle Daten angefordert wurden, bevor die nächste Methode ausgeführt wird. Ohne Promise würde die nächste Methode sofort ausgeführt werden, da _getArticles() asynchron ist. Diese Daten werden durch _toArray() in ein Array umgewandelt und _calcPrice() errechnet den Gesamtpreis.
Wenn der Nutzer auf den Buy-Button klickt, kommt die Methode onBuy() zur Ausführung. Darin werden die Kaufdaten in ein JSON-Format gebracht und an das Back-End geschickt. Die Anfrage ist ein XMLHttpRequest bzw. AJAX-Request. Mit der HTTP-Methode POST wird diese Anfrage an die URL http://localhost:4444/zcash/t/purchase gesendet. Davor werden die Header 'Authorization', token und 'Content-Type', 'application/json' gesetzt. Der Variable token beinhaltet einen Authentifizierungstoken. Wenn der Server den Status OK zurückgibt, wird der Warenkorb geleert und die Seite neu aufgebaut.

Listing 2.9: StoreBasket

```
class StoreBasket extends Polymer.Element {
  static get is() { return 'store-basket'; }

  static get properties() {
    return {
      basket: Array,
      sum: Number,
      decimals: {
        type: Number,
```

```
      value: 6,
    },
    orderComplete: Boolean,
  };
}
static get observers() {
  return [
    'basketChanged(basket.*)',
  ];
}
connectedCallback() {
  super.connectedCallback();
  this._renderPage();
}
ready() {
  super.ready();
  this.addEventListener('render-page', e => this._renderPage());
}
onBuy() {
  const _this = this;
  payload = JSON.stringify(payload);
  this._getToken().then(tokens => {
    const token = tokens[0].token;
    const xhr = new XMLHttpRequest();
    xhr.open('POST', 'http://localhost:4444/zcash/t/purchase', true);
    xhr.onload = function() {
      if (this.status === 200) {
        _this._removeArticles()
          .then(_this._renderPage());
        _this.orderComplete = true;
      }
    };
    xhr.setRequestHeader('Authorization', token);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.send(payload);
  });
}
basketChanged(item) {
  const article = this.basket[id].value.article;
  const name = article.name;
  // write to indexedDB
  idb.open('articleDB', 1, upgradeDB => {
    if (!upgradeDB.objectStoreNames.contains('articles')) {
      upgradeDB.createObjectStore('articles', {keyPath: 'name'});
    }
    }).then(db => {
    const tx = db.transaction('articles', 'readwrite');
    tx.objectStore('articles').put({name: name, article: article});
    return tx.complete;
  }).then(() => this._calcPrice());
}
_getToken() {
```

```
    return tokenPromise;
  }
  _renderPage() {
    this._getArticles()
    .then(basket => this._toArray(basket))
    .then(() => this._calcPrice());
  }
  _getArticles() {}
  _removeArticles() {}
  _toArray(obj) {}
  _calcPrice() {}
}
```

2.1.6 Webdesign

Das Webdesign bestimmt das Aussehen des Webshops. Die Anordnung der HTML-Elemente wurde im vorigen Abschnitt gezeigt. In diesem Abschnitt werden Farben, Abstände und Ausrichtung beschrieben. Dafür wird die Sprache CSS (Cascading Style Sheet) verwendet. CSS3 ist die aktuelle Version des Standards.
Der Webshop wurde nach den Grundsätzen Mobile First, Responsive Design und Mate-

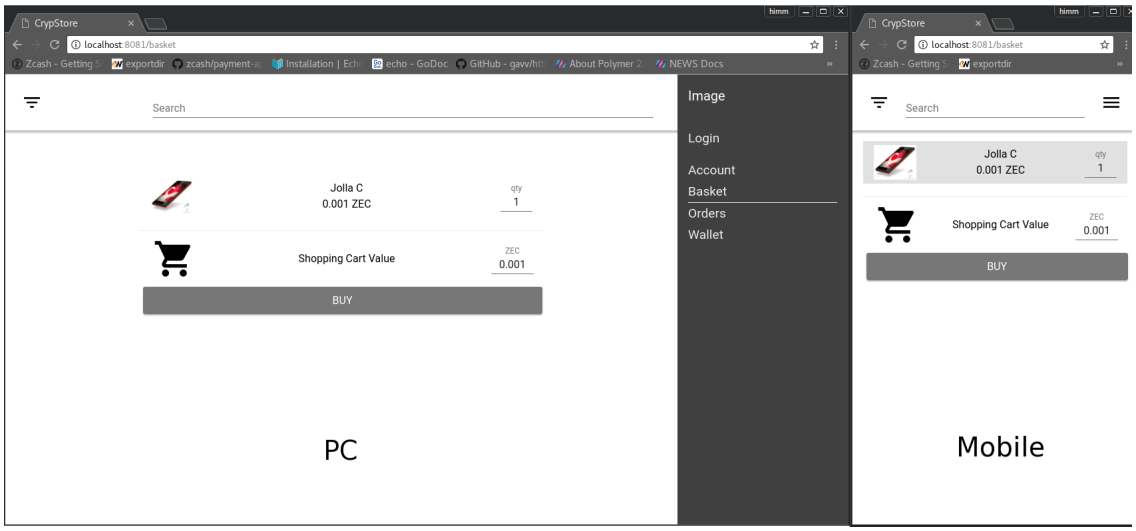

Abbildung 2.12: Warenkorb auf PC und Smartphone

rial Design entworfen. Mobile First besagt, dass die Elemente anhand eines Smartphones ausgerichtet werden. Für größere Bildschirme werden CSS-Regeln hinzugefügt. Responsive Design bedeutet, dass sich das Design an die Bildschirmgröße anpasst. Dafür gibt der Browser den Viewport an. Das ist i.d.R annähernd die Auflösung des Bildschirms. Wird die Größe das Browser-Fensters verändert, verändert sich der Viewport. An den Viewport werden Abstände und Anordnung von Elementen angepasst. Material Design ist eine Design-Richtlinie von Google. Alle Webkomponenten des Polymer-Teams wurde nach diesen Richtlinien entwickelt. Darin werden Schriftgrößen, Abstände

innerhalb von Elementen und Farbkombinationen vorgeschlagen.
Abbildung 2.12 zeigt den Webshop auf einem Desktop- (links) und Mobilgerät (rechts). Auf dem Smartphone kann der rechte Drawer über den Button in der rechten oberen Ecke geöffnet und geschlossen werden. Beim PC ist der Drawer dauerhaft offen, da genügend Platz vorhanden ist. Die Abstände des Contents zum Header und Drawer sind in der mobilen Version sichtlich geringer.
Die Implementierung wird nun anhand ausgewählter Beispiele beschrieben. Für jede Webkomponente gibt es eine große Anzahl von CSS-Regeln. In dieser Arbeit können nicht alle Details gezeigt werden. Die folgende Aufführung zeigt die Pseudoklasse :host der Shell. Darin wird zunächst der Display-Modus festgelegt. Der hier festgelegte Modus ist Standard in Chrom. Andere Browser nutzen display: inline-block; als Standard. Die explizite Angabe verhindert unerwünschte Effekte. Danach folgen vier CSS-Variablen. Sie wurden mit der Version 2 eingeführt und in Version 3 aktualisiert. CSS-Variablen werden durch den Präfix „–“ gekennzeichnet. Hier werden die vier Hauptfarben des Webshops definiert.

Listing 2.10: <store-shell> :host

```
:host {
  display: block;
  --primary-light-color: #fff;
  --secondary-light-color: #eee;
  --primary-dark-color: #404040;
  --secondary-dark-color: #777;
}
```

<app-drawer> ist ein Polymer-Element. Es wird als ein gewöhnliches HTML-Element gehandelt. Der Name des Elements wird ohne Präfix aufgeführt. In der ersten Zeile wird die Farbe des Textes festgelegt. Der Inhalt der Variable –primary-light-color wird in der Form var(Variable) ausgelesen. Danach folgen zwei Variablen, die nicht in der Shell definiert sind. Sie sind Teil des <app-drawer>. Darin wird die Hintergrundfarbe der Seite beim geöffneten Drawer festgelegt. Die zweite Variable definiert die Hintergrundfarbe des Drawers und einen Schatten.

Listing 2.11: <store-shell> app-drawer

```
app-drawer {
  color: var(--primary-light-color);
  --app-drawer-scrim-background: rgba(0, 0, 0, 0.05);
  --app-drawer-content-container: {
    background-color: var(--primary-dark-color);
    box-shadow: 2px 0 2px 0 rgba(0,0,0,0.18);
  };
}
```

Flexbox ein neuer Displaymodus, welcher in CSS3 eingeführt wurde. Damit werden Elementen anhand des zur Verfügung stehenden Platzes angeordnet. Die CSS-Eigenschaft

flex-wrap gibt an, dass überlaufende Elemente in die nächste Zeile verschoben werden. So werden Artikel auf dem Smartphone in einer Spalte angezeigt. Auf dem Desktop-PC sind mehrere Artikel in einer Zeile. Die Anzahl hängt vom Viewport ab.

Listing 2.12: <store-smartphones> .container

```
.container {
  display: flex;
  flex-wrap: wrap;
}
```

Media-Queries wurden ebenfalls mit CSS2 eingeführt und in Version 3 aktualisiert. Sie ermöglichen das Anpassen von CSS-Regeln anhand der Viewport-Größe.

Listing 2.13: <store-smartphones> @media

```
@media (min-width: 480px) {
  :host {
    padding: 1rem 1rem 0 1rem;
  }
}
```

2.2 Das Back-End

Das Back-End des Webshops wurde mit der Programmiersprache Go implementiert. Go ist hervorragend für diese Aufgabe geeignet. Es ist eine sehr performante Sprache, gleichzusetzen mit C++. Der Vorteil gegenüber C++ liegt in der Einfachheit der Sprache. Außerdem benötigt Go erheblich weniger Zeit zum kompilieren. Das beschleunigt den Entwicklungsprozess. Für Go gibt es einige Frameworks zur Webentwicklung. In diesem Projekt wurde Echo gewählt. Es bringt viele nützliche Funktionen mit, die das Entwickeln vereinfacht und schöneren Syntax ermöglicht. Das Back-End beinhaltet einen eigenen Webserver. Er nimmt die Anfragen des Clients entgegen und gibt eine Antwort zurück. In dieser Arbeit können nicht alle Funktionen des Back-Ends gezeigt werden.

2.2.1 Aufbau

Das Back-End ist modular aufgebaut. D.h. es wurde in logisch zusammenhängende Module aufgeteilt. Go unterteilt den Quellcode in Pakete (package). Jedes Paket muss einem eigenen Order haben. Enthält ein Ordner Dateien mit verschiedenen Paketdeklarationen, gibt der Compiler einen Error zurück. Diese Pakete sind account, models, mongo, orders, router, session, user, validator, wallet und zcash. Eine Ausnahme bildet die Startdatei main.go. Sie gehört immer zum Paket main. Die Sichtbarkeit von Variablen und Funktionen wird auf Basis des Pakets definiert.

```
account/
  handler.go
  handler_test.go
Dockerfile
main.go
models/
  account.go
  mongo.go
mongo/
  account.go
  handler.go
  orders.go
  user.go
  wallet.go
orders/
  handler.go
router/
  account.go
  orders.go
  router.go
  session.go
  user.go
  wallet.go
  zcash.go
session/
  claims.go
  handler.go
  rand.go
user/
  handler.go
  handler_test.go
validator/
  user.go
wallet/
  handler.go
  handler_test.go
zcash/
  handler.go
  transaction.go
```

Abbildung 2.13: Aufbau des Back-Ends

Account enthält Funktionen, um einen Account anzulegen, oder zu löschen. Models beinhaltet alle Strukturen bzw. Modelle des Back-Ends. Mongo dient als Schnittstelle zwischen Webserver und der MongoDB-Datenbank. Orders definiert Funktionen, um Bestellungen zu persistieren. Router stellt das Routing dar. Anfragen an den Webserver werden hier verteilt. Session beinhaltet Funktionalität zur Sitzungsverwaltung. User erstellt und überschreibt Nutzerdaten. Validator enthält Funktionen zur Eingabevalidierung. Wallet erstellt, verwaltet und löscht Wallets. Ein Wallet ist eine Art Geldbörse für Kryptowährungen. Das Paket zcash implementiert Funktionen zum Senden von und Bezahlen mit Zcash.

Die meisten Pakete enthalten eine Datei handler.go. Sie enthält alle Funktionen des Pakets, welche der Webserver direkt aufruft. In den folgenden Abschnitten wird die Implementierung der wichtigsten Pakete genau erklärt.

2.2.2 Die Startdatei main.go

Die Datei main.go ist der Startpunkt jedes Go-Programms. Aus ihr werden die Funktionen anderer Pakete aufgerufen.
Jede Go-Datei beginnt mit dem Paketnamen (package).

```
package main
```

Das Paket main enthält nur die Datei main.go. Nach der Paketangebe folgen die Imports. Andere Pakete werden importiert, um dessen Funktionen aufzurufen. Die Pakete mongo und router wurden vom Autor implementiert. Echo, middleware und autocert sind Pakete des Echo-Frameworks. In den folgenden Abschnitten werden package und import nicht angegeben.
Eine Funktion wird in Go mit dem Wort func gekennzeichnet. Generell haben Funktionen den folgenden Aufbau.

```
func (t *struct) name(args...) (returnVals...) {}
```

Go beinhaltet keine Klassen. Stattdessen werden Strukturen definiert, auf welche Funktionen angewandt werden. Diese heißen Methoden. Die zugehörige Struktur wird nach func angegeben. Danach folgt der Name der Funktion/Methode und dessen Argumente. Zuletzt werden die Rückgabewerte angegeben. Funktionen können beliebig viele Werte zurückgeben. Funktionen und Variablen, die mit einem großen Buchstaben beginnen, werden aus dem Paket exportiert (exported functions/varaibles). Sie sind somit außerhalb des Pakets sichtbar.
Als Beispiel wird wird eine Struktur Word angegeben. Deren Methode Capitalize, wird s als Typ string übergeben. Sie gibt eine Zeichenkette und einen Error zurück. Darunter folgt der Aufruf der Methode.

Listing 2.14: Beispiel

```
type Word struct {
    Name string
}

func (w *Word) Capitalize(s string) (string, error) {
    w.Name = strings.ToUpper(s[:1])
    w.Name += (s[1:])
    return w.Name, nil
}

w := Word{}
```

```
s, _ := w.Capitalize("a word")
```

In der Funktion main wird zuerst eine Instanz e vom Paket echo erstellt. Danach wird die Middleware des Programms festgelegt. Middleware sind Funktionen, die vor dem Aufruf einer anderen Funktion ausgeführt werden. Sie werden sozusagen zwischen dem Aufruf der Funktion und dessen Durchführung ausgeführt. Für dieses Projekt wurde Logger, Recover und CORSWithConfig gewählt. Logger loggt alle Anfragen und Antworten des Webservers. Recover stellt sicher, dass der Webserver weiterläuft, wenn ein Fehler auftritt. CORSWithConfig ist eine Middleware, die das Cross Origin Resource Sharing (CORS) vereinfacht. Hier sind alle GET-, PUT,- POST- und DELETE-Anfragen der von http://localhost:8081 erlaubt. Als nächstes wird eine Verbindung zu MongoDB aufgebaut und Indexe zur der Datenbank hinzugefügt. Zuletzt werden die Routen festgelegt und der Webserver gestartet.

Listing 2.15: main.go

```
package main

import (
  "crypstore/backend/mongo"
  "crypstore/backend/router"

  "github.com/labstack/echo"
  "github.com/labstack/echo/middleware"
  "golang.org/x/crypto/acme/autocert"
)

func main() {
  // init echo
  e := echo.New()
  // middelware
  e.Use(middleware.Logger())
  e.Use(middleware.Recover())
  e.Use(middleware.CORSWithConfig(middleware.CORSConfig{
      AllowOrigins: []string{"http://localhost:8081"},
      AllowMethods: []string{echo.GET, echo.PUT, echo.POST, echo.DELETE},
  }))
  // init mongo
  mongo.CreateSession()
  mongo.AddIndex()
  // start server
  router.SetRoutes(e)
  e.Logger.Fatal(e.Start(":4444"))
}
```

2.2.3 Das Paket router

Der Router wird aus der Datei main.go aufgerufen. Er legt die Routen des Webservices fest. Die Datei router.go ist der Eintrittspunkt des Pakets. Aus ihr werden alle Dateien aufgerufen, die das Routing enthalten. Das sind die Pakete account, user, session, orders, wallet und zcash. Für jedes Paket gibt es eine eigene Datei. Jeder Datei wird ein Zeiger auf die Instanz des Echo-Frameworks übergeben.

Listing 2.16: router/router.go

```
func SetRoutes(e *echo.Echo) {
  setAccountRoutes(e)
  setUserRoutes(e)
  setSessionRoutes(e)
  setOrdersRoutes(e)
  setWalletRoutes(e)
  setZcashRoutes(e)
}
```

Eine dieser Dateien ist account.go. Sie ist im Paket router enthalten. Die Routen werden in zugänglich (accessible) und beschränkt (restricted) unterteilt. In account.go wird zunächst eine zugängliche Route festgelegt. Bei jedem Aufruf des Pfades /account mit der HTTP-Methode POST wird die Funktion Create des Pakets account aufgerufen. Danach folgt eine beschränkte Route. Sie implementiert die Middleware JWT. Durch JSON Web Tokens authentifizieren sich Nutzer am Back-End. Somit können nur eingeloggte Nutzer die Methode account.Delete aufrufen. Dieser Aufruf erfolgt durch eine DELETE-Anfrage an /account.

Listing 2.17: router/account.go

```
func setAccountRoutes(e *echo.Echo) {
  // accessible
  e.POST("/account", account.Create)
  // restricted
  e.DELETE("/account", account.Delete, middleware.JWT(session.SignKey))
}
```

2.2.4 Das Paket models

In models sind alle Strukturen (struct) enthalten. Das Paket enthält die zwei Dateien mongo.go und account.go. Mongo.go enthält eine Struktur, welche die Konfiguration der Datenbank enthält. Account.go enthält die vier Strukturen Account, User, Order und Wallet.
Account ist dabei die übergeordnete Struktur. Sie beinhaltet die anderen drei Strukturen. Zudem is ein Feld Id von Typ bson.ObjectId enthalten. MongoDB kann Documente schneller suchen, wenn die Einträge indiziert sind. Id ist ein Index, welcher aufgrund

von User.Name gebildet wird. In einer Go-Struktur werden Namen für andere Datenformate hinter dem Datentyp angegeben. Beispielsweise enthält die Struktur User eine Zeichenkette Name. Sie wird in der Datenbank (bson), als JSON und in einer Form als name angegeben. Ein Feld wird nicht übertragen, falls omitempty gesetzt ist.

```
Name string `bson:"name" json:"name,omitempty" form:"name,omitempty"`
```

Die Struktur User enthält den Namen, das Passwort, die Email-Adresse und die Lieferadresse des Nutzers. Order stellt einen gekauften Artikel dar.
Dabei werden Datum, Name des Artikels und die Anzahl gespeichert. In Account ist ein Slice von Order ([]Order) im Feld Orders gespeichert. Ein Slice ähnelt einem Array, kann jedoch performanter implementiert und erweitert werden.
Das Wallet enthält den öffentlichen und privaten Zcash-Schlüssel. Der private Schlüssel wird nie als JSON oder Form übertragen. Das wird durch „-“ sichergestellt. Außerdem enthält es das Guthaben des Nutzers.

Listing 2.18: models/account.go

```
type (
  // Account represents a user account
  Account struct {
    Id      bson.ObjectId `bson:"_id,omitempty" json:"-" form:"-"`
    User    User          `bson:"user" json:"user,omitempty" form:"user,
        omitempty"`
    Orders  []Order       `bson:"orders" json:"orders,omitempty" form:"orders
        ,omitempty"`
    Wallet  Wallet        `bson:"wallet" json:"wallet,omitempty" form:"wallet
        ,omitempty"`
  }

  // User represents user data
  User struct {
    Name     string `bson:"name" json:"name,omitempty" form:"name,omitempty"`
    Pass     string `bson:"pass" json:"pass,omitempty" form:"pass,omitempty"`
    Email    string `bson:"email" json:"email,omitempty" form:"email,
        omitempty"`
    Address string `bson:"address" json:"address,omitempty" form:"address,
        omitempty"`
  }

  // Order represents a order
  Order struct {
    Date string `json:"date,omitempty" from:"date,omitempty"`
    Name string `json:"name,omitempty" from:"name,omitempty"`
    Qty  string `json:"qty,omitempty" from:"qty,omitempty"`
  }

  // Wallet represents a wallet for zcash
  Wallet struct {
```

```
    Zaddr      string   `json:"taddr,omitempty" form:"taddr,omitempty"`
    ZaddrPriv  string   `json:"-" form:"-"`
    Zbalance   float64  `json:"tbalance" form:"tbalance,omitempty"`
  }
)
```

2.2.5 Das Paket account

Das Paket Account beinhaltet Funktionen zum Erstellen und Löschen einen Benutzerkontos. Create() erstellt ein Konto und Delete() löscht ein Konto. Funktionen, die von Router aufgerufen werden, heißen Handler. Einer Handlerfunktion wird der Context der HTTP-Pakete (echo.Context) übergeben. Dies ist ein Schnittstelle (interface) des Echo-Frameworks. Sie enthält unter anderem die Funktionen Request, Response, FormValue, JSON und Error. Dabei stellt Request eine HTTP-Anfrage dar. Response ist eine HTTP-Antwort. Mit FormValue() können Daten einer Anfrage gelesen werden, die als Form übertragen wurden. Die Funktion JSON() gibt ein Response mit einem Statuscode und einem JSON zurück. Error() gibt einen generierten Fehlercode und eine Fehlernachricht zurück.
Die Funktion Create bekommt eine Form als Context übergeben. Sie enthält den Namen, das Passwort, die Email-Adresse und die Lieferadresse des Nutzers. Diese Daten werden in eine zuvor erstellte Instanz von models.Account gespeichert. Danach folgt die Validierung der Eingabe. Sollte ValidateUser einen Fehler zurückgeben, bekommt der Client eine Antwort mit dem Fehlercode 400 (Bad Request) und die Nachricht „user data invalid “. Andernfalls werden die validierten Daten in die Datenbank geschrieben. Dazu wird die Funktion CreateAccount des Pakets mongo ausgeführt. Der Ausdruck defer definiert eine Funktion, die immer nach der darüberliegenden Funktion ausgeführt wird. Das ist in diesem Fall Create(). Die Funktion s.Close() beendet die Datenbanksitzung. Sie wird auch ausgeführt, wenn sich das Programm fehlerhaft beendet. Falls es keine Fehler gab, gibt Create() den Status 201 (Created) zurück.

Listing 2.19: account/handler.go - Create

```
func Create(ctx echo.Context) error {
  a := models.Account{}
  a.User.Name = ctx.FormValue("name")
  a.User.Pass = ctx.FormValue("pass")
  a.User.Email = ctx.FormValue("email")
  a.User.Address = ctx.FormValue("address")
  if err := validator.ValidateUser(&a.User); err != nil {
    return ctx.JSON(http.StatusBadRequest, "user data invalid")
  }
  s, err := mongo.CreateAccount(&a)
  defer s.Close()
  if err != nil {
    return ctx.NoContent(http.StatusForbidden)
  }
```

```
  return ctx.NoContent(http.StatusCreated)
}
```

Delete() bekommt den Namen des Nutzers als JWT-Claim übergeben. Dazu mehr im folgenden Kapitel. Durch den JSON Web-Token ist der Nutzer authentifiziert. Der Account mit dem Nutzernamen wird gelöscht und der Statuscode 200 (OK) zurückgegeben.

Listing 2.20: account/handler.go - Delete

```
func Delete(ctx echo.Context) error {
  name := session.GetClaimsName(ctx)
  s, err := mongo.DeleteAccount(name)
  defer s.Close()
  if err != nil {
    return ctx.NoContent(http.StatusNotFound)
  }
  return ctx.NoContent(http.StatusOK)
}
```

2.2.6 Das Paket session

Session ist das Paket, welches Funktionen zur Sitzungsverwaltung (session management) enthält. Bevor ein Nutzer beschränkte Funktionen aufrufen kann, muss er sich am Server authentifizieren. Die Authentifizierung erfolgt durch einen JSON Web Token (JWT). Dabei werden Daten des Nutzers (Claims) mit einem privaten Schlüssels signiert. Der so entstandene Token passt nur auf den Nutzer. Es ist kein Rückschluss auf den privaten Schlüssel des Servers möglich.
Zum Authentifizieren wird die Funktion Login() aufgerufen. Der Client übergibt das den Namen und das Passwort. Das Passwort wird nicht als Hash übertragen. Diese Implementierung ist dementsprechend unsicher. Danach wird das Passwort mit dem in der Datenbank hinterlegten Passwort verglichen. Falls diese übereinstimmen, wird ein JWT generiert und an den Client gesendet. Der Token wird aufgrund des Nutzernamens gebildet und ist in diesem Fall drei Tage gültig. Der Client löscht den Token jedoch beim Ausloggen. Stimmt das Passwort nicht, gibt der Server den Statuscode 403 (Unauthorized) zurück.

Listing 2.21: session/handler.go - Login

```
func Login(ctx echo.Context) error {
  name := ctx.FormValue("name")
  pass := ctx.FormValue("pass")
  if name == "" {
    log.Printf("session > Login > name: %s\n")
    return ctx.NoContent(http.StatusBadRequest)
  }
  u, s, err := mongo.GetUser(name)
  defer s.Close()
```

```
  if err != nil {
    return ctx.JSON(http.StatusInternalServerError, "user not found")
  }
  if u.Pass == pass {
    token := jwt.New(jwt.SigningMethodHS256)
    // set claims
    claims := token.Claims.(jwt.MapClaims)
    claims["name"] = u.Name
    claims["exp"] = time.Now().Add(time.Hour * 72).Unix()
    // generate encoded token
    tokenStr, err := token.SignedString(SignKey)
    if err != nil {
      return ctx.JSON(http.StatusInternalServerError, "JWT not signed")
    }
    return ctx.JSON(http.StatusOK, map[string]string{
      "token": tokenStr,
    })
  }
  return ctx.NoContent(http.StatusUnauthorized)
}
```

Die Funktion GetClaimsName() wird von allen Funktionen aufgerufen, welche eine Authentifizierung erfordern. Sie liest den Nutzernamen aus den Claims und gibt ihn zurück.

Listing 2.22: session/claims.go - GetClaimsName

```
func GetClaimsName(ctx echo.Context) string {
  user := ctx.Get("user").(*jwt.Token)
  claims := user.Claims.(jwt.MapClaims)
  name := claims["name"].(string)
  return name
}
```

2.2.7 Tests

Go bringt das Paket testing mit, welches das Testen von Funktionen ermöglicht. Diese Tests rufen Funktionen direkt auf. In diesem Projekt wurde stattdessen das Paket httpexpect verwendet. Es baut auf testing auf. Im Gegensatz zu testing wird ein Webserver gestartet und „echte“ Anfragen versendet. Mit dem Befehl go test im Verzeichnis des zu testenden Paktes werden die Tests gestartet.

```
$ cd account
$ go test
```

Eine Testmethode beginnt mit „Test“, gefolgt vom Namen der Funktion. Httpexpect besitzt nur die Methode TestEchoClient(). Daraus werden alle anderen Testfunktionen aufgerufen. Der Handler für das benutzte Framework wird zuerst instanziert. Danach wird

der Webserver gestartet. Daraus wird eine Instanz von httpexpect gebildet. Diese Instanz wird den Testfunktionen übergeben. Das folgende Beispiel ruft die Testfunktion testCreate() auf. Sie testet die Funktion Create() des Pakets account.

Listing 2.23: account/handler_test.go - TestEchoClient

```
func TestEchoClient(t *testing.T) {
  // setup
  handler := EchoHandler()
  server := httptest.NewServer(handler)
  defer server.Close()
  e := httpexpect.New(t, server.URL)
  // tests
  testCreate(e)
}

func EchoHandler() http.Handler {
  e := echo.New()
  // set routes
  e.POST("/account", Create)
  // mongo
  mongo.CreateSession()
  mongo.AddIndex()
  return e
}
```

Für diesen Test wurde zunächst eine Struktur erstellt, welche Testdaten eines Nutzers enthält. Die Funktion teardown() wird nach dem Test ausgeführt. Sie löscht den Account aus der Datenbank. Danach wird eine POST-Anfrage an den Webserver geschickt. Die Anfrage enthält die Testdaten als Form. Falls der Server mit dem Status Created antwortet, war der Test erfolgreich.

Listing 2.24: account/handler_test.go - testCreate

```
var (
  u = models.User{
    Name:    "test",
    Pass:    "hash1234",
    Email:   "test@tester.com",
    Address: "Weg 1, 0000 Dorf",
  }
)

func testCreate(exp *httpexpect.Expect) {
  defer teardown()
  exp.POST("/account").WithForm(u).
    Expect().Status(http.StatusCreated)
}
```

2.3 Zahlungsabwicklung mit Zcash

Die Zahlungsabwicklung mit Zcash ist ein Teil des Back-Ends. Dafür wurden die Pakete wallet und zcash implementiert. Um Transaktionen mit Zcash durchzuführen, muss zuerst der Zcash-Client (zcashd) gestartet werden. Zcash bringt eine eigene API mit, um Transaktionen und Walletdaten zu verwalten. Die wichtigsten Bestandteile der Implementierung werden nun vorgestellt.

2.3.1 Zcash API

Zcash basiert auf dem Bitcoin-Protokoll. Transparente Adressen können die API-Aufrufe des Bitcoin-Protokolls nutzen. Die API von Zcash erweitert die Bitcoin-API um Befehle, welche auf geschützte Adressen angewandt werden. Diese Befehle beginnen mit dem Präfix „z_“. Sie können oft zusätzlich transparente Adressen nutzen.

Listing 2.25: Befehle

```
z_exportkey
z_exportwallet
z_getbalance
z_getnewaddress
z_getoperationresult
z_getoperationstatus
z_gettotalbalance
z_importkey
z_importwallet
z_listaddresses
z_listoperationids
z_listreceivedbyaddress
z_sendmany
z_validateaddress
```

Um die Zcash-API zu nutzen, muss zuerst der Client gestartet werden. Dazu wird der Befehl zcashd ausgeführt. Die Option -daemon veranlasst das Starten als Hintergrundprozess. Danach können Befehle durch zcash-cli, gefolgt vom Befehl, aufgerufen werden.

```
$ zcashd -daemon
$ zcash-cli command
```

In den folgenden zwei Kapiteln werden die Befehle erläutert, welche in dieser Arbeit benutzt wurden.

2.3.2 Das Paket wallet

Das Paket wallet implementiert Funktionen zum Erstellen, Lesen und Löschen eines Wallets. Ein Wallet ist eine Art Geldbörse für Kryptowährungen. In diesem Fall enthält das Wallet nur eine geschützte Zcash-Adresse. Es kann jedoch beliebig viele transparente und geschützte Adressen enthalten.
Die Funktion Create holt zunächst den Nutzernamen aus den Claims. Dafür muss der Nutzer eingeloggt sein. Dann wird ein Instanz der Struktur Wallet erstellt. Danach wird durch den API-Aufruf zcash-cli z_getnewaddress eine neue geschützte Adresse erstellt. Das ist der öffentliche Schlüssel. Sie wird in das Feld Zaddr der Wallet-Instanz geschrieben. Als nächstes folgt der API-Aufruf zcash-cli z_exportkey. Er bekommt die zuvor erstellte Adresse aus Parameter übergeben. Der Aufruf gibt den privaten Schlüssel zurück. Zuletzt werden die Schlüssel in der Datenbank hinterlegt. Wenn keine Fehler aufgetreten sind, gibt die Funktion den Status Created zurück.

Listing 2.26: wallet/handler.go - Create

```
func Create(ctx echo.Context) error {
  name := session.GetClaimsName(ctx)
  w := models.Wallet{}
  out, err := exec.Command("zcash-cli", "z_getnewaddress").Output()
  if err != nil {
    log.Printf("wallet > Create > z_getnewaddress: %s\n", err)
    return err
  }
  sOut := string(out)
  w.Zaddr = strings.TrimSuffix(sOut, "\n")
  out, err = exec.Command("zcash-cli", "z_exportkey", w.Zaddr).Output()
  if err != nil {
    log.Printf("wallet > Create > z_exportkey: %s\n", err)
    return ctx.JSON(http.StatusInternalServerError, "Could not get private
        key")
  }
  sOut = string(out)
  w.ZaddrPriv = strings.TrimSuffix(sOut, "\n")
  s, err := mongo.SetWallet(name, &w)
  defer s.Close()
  if err != nil {
    return ctx.JSON(http.StatusInternalServerError, "Could not set wallet")
  }
  return ctx.JSON(http.StatusCreated, w)
}
```

Get gibt des Walletdaten eines Nutzers zurück. Dafür wird zuerst der Name aus den Claims gelesen. Danach wird der Befehl zcash-cli z_getbalance ausgeführt. Die Zcash-Adresse des Nutzers wird dafür aus der Datenbank gelesen und als Parameter übergeben. Zuletzt werden Guthaben (balance) und Adresse als JSON an den Client gesendet.

Listing 2.27: wallet/handler.go - Get

```
func Get(ctx echo.Context) error {
  name := session.GetClaimsName(ctx)
  w, s, err := mongo.GetWallet(name)
  defer s.Close()
  if err != nil {
    return ctx.NoContent(http.StatusNotFound)
  }
  out, err := exec.Command("zcash-cli", "z_getbalance", w.Zaddr).Output()
  if err != nil {
    log.Printf("wallet > Read > amount: %s\n", err)
    return ctx.JSON(http.StatusInternalServerError, "Could not get balance")
  }
  sOut := string(out)
  w.Zbalance, _ = strconv.ParseFloat(strings.TrimSuffix(sOut, "\n"), 64)
  return ctx.JSON(http.StatusOK, w)
}
```

Delete löscht das Wallet des Nutzers. Dabei wird der Eintrag in MongoDB mit einem leeren Eintrag überschreiben. Für das Löschen muss sich der Nutzer zuvor authentifiziert haben. Wenn der Löschvorgang erfolgreich was, gibt der Server den Status OK zurück.

Listing 2.28: wallet/handler.go - Delete

```
func Delete(ctx echo.Context) error {
  name := session.GetClaimsName(ctx)
  s, err := mongo.DeleteWallet(name)
  defer s.Close()
  if err != nil {
    return ctx.NoContent(http.StatusNotFound)
  }
  return ctx.NoContent(http.StatusOK)
}
```

2.3.3 Das Paket zcash

In zcash wurden Handlerfunktionen für das Bezahlen und Senden von Zcash implementiert. Diese Implementierung funktioniert nicht für transparente Adressen. Beim Senden von Zcash, wird kein Guthaben als Zahl überwiesen. Das Bitcoin- und Zcash-Protokoll beinhaltet keine Guthaben im herkömmlichen Sinn. Technisch gesehen hat eine Adresse Input-Transaktionen (TxIn) und Output-Transaktionen (TxOut). Die Differenz zwischen TxIn und TxOut bildet das Guthaben. Es können jedoch nur TxIn als TxOut versendet werden. Wenn eine Adresse nur ein TxIn von 1 ZEC enthält und 0,7 ZEC versendet möchte, kann der Protokoll nur den TxIn von 1 ZEC senden. Dabei wird die Transaktion aufgeteilt und die übrigen 0,3 ZEC einer neuen Adresse hinzugefügt. Diese 0,3 ZEC müssen dann zur ursprünglichen Adresse zurückgesendet werden. Das

Zcash-Protokoll implementiert dieses Verhalten für transparente Adressen, um die Anonymität des Nutzers zu schützen. Das Rücksenden an transparente Adressen wurde nicht implementiert. Bei geschützten Adressen ist das nicht notwendig. Überschüssiges Guthaben wird automatisch zum Sender zurückgeschickt.
Die Funktion Purchase beinhaltet die Zcash-Adresse des Webshops. Sie wurde in dieser Arbeit nicht vollständig angegeben. Alle Überweisungen eines Einkaufs werden an diese Adresse versandt. Der Nutzer wird durch einen JWT authentifiziert. Die Bestellung wird vom Client als JSON verschickt. Sie wird in der Struktur gespeichert. Danach wird das Wallet des Nutzers geladen. Von seiner Zcash-Adresse wird der in getPrice berechnete Preis versendet. Der folgende Ausschnitt zeigt die Implementierung. Der Befehlt zcash-cli z_sendmany wird ausgeführt. Der erste Parameter ist die Senderadresse . Danach folgt ein Array im JSON-Format, welches die Empfängeradresse und den Betrag enthält. Der letzen zwei Paramenter geben an, dass eine Transaktion vier mal bestätigt werden soll und keine Transaktionsgebühren berechnet werden.

Listing 2.29: zcash/transaction.go - z_sendmany

```
if balance >= amount && amount > 0 {
  a := fmt.Sprintf("[{\"address\": \"%s\", \"amount\": %.8f}]", dst, amount)
  cmd := exec.Command("zcash-cli", "z_sendmany", w.Zaddr, a, "4", "0")
```

Dabei wird geprüft, ob der Nutzer ausreichend Guthaben besitzt. Wenn die Bestellung erfolgreich war, wird Sie in der Datenbank hinterlegt. Zuletzt gibt der Server den Status OK zurück.

Listing 2.30: zcash/handler.go - Purchase

```
func Purchase(ctx echo.Context) error {
  var dst = "zcdyT6b5iJSEftCrCNt98fyZCmXxFvvug4T1***"
  name := session.GetClaimsName(ctx)
  orders := []models.Order{}
  err := ctx.Bind(&orders)
  if err != nil {
    log.Printf("zcash > Pay > Bind: %s\n", err)
    return err
  }
  w, s, err := mongo.GetWallet(name)
  defer s.Close()
  if err != nil {
    return ctx.NoContent(http.StatusNotFound)
  }
  amount := getPrice(&orders)
  err = sendFunds(w, dst, amount, ctx)
  if err != nil {
    return err
  }
  // set orders
  s, err = mongo.AddOrders(name, &orders)
  if err != nil {
    return ctx.JSON(http.StatusInternalServerError, "Order not added")
```

```
  }
  return ctx.NoContent(http.StatusOK)
}
```

Send hat eine ähnliche Funktionalität wie Purchase. Im Gegensatz zu Purchase wird ihr die Empfängeradresse übergeben. Es werden keine Bestellungen in der Datenbank hinterlegt.

Listing 2.31: zcash/handler.go - Send

```
func Send(ctx echo.Context) error {
  name := session.GetClaimsName(ctx)
  amount, _ := strconv.ParseFloat(ctx.FormValue("amount"), 64)
  dst := ctx.FormValue("dst")
  w, s, err := mongo.GetWallet(name)
  defer s.Close()
  if err != nil {
    return ctx.NoContent(http.StatusNotFound)
  }
  // send funds
  err = sendFunds(w, dst, amount, ctx)
  if err != nil {
    log.Printf("zcash > Pay > sendFunds: %s\n", err)
    return err
  }
  return ctx.NoContent(http.StatusOK)
}
```

3 Ergebnisse

Ein Webshop mit Zcash-Zahlungsfunktionalität konnte implementiert werden.
Das Front-End wurde mit dem Framework Polymer 2 entwickelt. Es beinhaltet Artikel, welche zur Demonstration dienen. Seiten zum Einloggen und Registrieren wurden implementiert. Zudem gibt es eine Übersicht der Nutzerdaten. Diese können überschrieben werden. Der Warenkorb verwaltet Artikel in einer IndexedDB. Sie sind nicht von der Sitzung abhängig und bleiben nach dem Browserneustart erhalten. Vergangene Bestellungen sind einsehbar. Außerdem enthält der Webshop ein Wallet, welches die öffentliche Zcash-Addresse und desses Guthaben anzeigt. Es wurde eine Shell implementiert, welche zunächst das Layout und später den Content lädt. Dabei wurden zwei Drawer und ein Header implementiert. Die Drawer dienen zur Navigation. Der Header beinhaltet eine Suchleiste. Die Suchfunktion wurde nicht implementiert. Der Content der Seite wird asynchron geladen. Für die Entwicklung des Front-Ends wurden sehr aktuelle Webtechnologien verwendet. Durch den Einsatz von Shadow DOM konnten Webkomponenten implementiert werden. Sie werden performant und asynchron geladen. Im Webshop sind keine DOM-blockierenden Elemente enthalten. Der Service Worker ermöglicht das offline Cachen und Laden des Contents. Der Webshop wurde vollständig nach dem Prinzipien responsive Webdesign und Mobile First entwickelt. Die Webkomponenten basieren auf Googles Material Design.
Für die Entwicklung des Back-Ends wurde das Framework Echo benutzt. Es hat einen modularen Aufbau. Die Implementierung ist sehr performant. Der integrierte Webserver enthält Logging-, Recover- und CORS-Funktionalität. Ein Router nimmt Anfragen entgegen und ruft die gewünschte Funktion auf. Das Paket account ermöglicht das Erstellen und Löschen von Benutzerkonten. Funktionen zum Lesen und Überschreiben der Nutzerdaten wurden implementiert. Durch das Datenbanksystem MongoDB werden diese persistiert. Der Webservice ermöglicht eine Sitzungsverwaltung aufgrund von JSON-Web-Tokens. Außerdem enthält das Back-End eine Zahlungsfunktionalität mit der Kryptowährung Zcash. Das Guthaben kann gelesen und versendet werden.

4 Zusammenfassung

In dieser Arbeit wurde ein Webshop implementiert. Dieser besteht aus einem Front-End und einem Back-End. Der Webshop bietet eine Zahlungsfunktionalität mit der Kryptowährung Zcash. Er wurde von Grund auf und ohne CMS entwickelt.
Es wurde gezeigt, wie ein Front-End mit dem Framework Polymer 2 erstellt werden kann. Dabei wurde das Erstellen eines Polymer-Projekts erklärt. Das Entwickeln von Webkomponenten wurde im Detail beschrieben. Die Implementierung wurde anhand des Quellcodes erläutert. Die Webtechnologien Shadow DOM, Webkomponenten, Service Worker und IndexedDB wurden im Detail vorgestellt. Das Webdesign wurde in Grundzügen beschrieben.
Das Back-End wurde in der Programmiersprache Go und mit dem Framework Echo geschrieben. Der modulare Aufbau wurde gezeigt. Die Implementierung des Webservice wurde anhand des Quellcodes erklärt. Die Zahlungsfunktionalität mit Zcash ist Teil des Back-Ends.
Ein Nutzer kann Waren in den Warenkorb legen, sich einloggen und bestellen. Er kann seine Daten ändern. Eine erfolgreiche Bestellung wird angezeigt. Gekaufte Artikel und das Zcash-Guthaben werden angezeigt.

5 Diskussion

Ein Webshop, der die Möglichkeit bietet mit Kryptowährungen zu bezahlen, ist ein Schritt in die richtige Richtung. Er ermöglicht es Personen, welche bereits Kryptowährungen besitzen, diese zu nutzen. Zudem erkennen Neueinsteiger den Wert dieser Währungen besser, wenn sie ein Produkt daran messen können. Der Webshop dieser Arbeit ist ein Proof of Concept. Er zeigt, dass es möglich ist, eine Zahlungsabwicklung mit der Kryptowährung Zcash zu implementieren. Die Implementierung des Webshop hat jedoch viele Schwachstellen und bietet Platz für Verbesserungen. Zudem ermöglicht ein Webshop allein nicht den Wandel von Fiat- zu Kryptowährungen. Neben den politischen Aspekten, muss die Akzeptanz dieser Währungen vergrößert und das Image gestärkt werden.

5.1 Implementierung

Der Webshop nutzt einige der neusten Webtechnologien. Diese können sicher besser implementiert werden. Zudem gibt es einige Technologien, welche nicht genutzt wurden. Dazu gehört z.B. die fetch API, welche ein neuer Standard ist um HTTP-Anfragen zu bilden. Anfragen wurden in dieser Arbeit durch AJAX implementiert. Das Deployment des Webshops kann auf unterschiedliche Arten durchgeführt werden. Diese würden sich in Performance und Kosten unterscheiden.
Die Shell des Front-Ends kann weiter ausgelagert werden, um bessere Ladezeiten zu ermöglicen. Die Datei manifest.json ist ein wichtiger Bestandteil von Progressive Web Apps (PWA). Sie wurde nicht konfiguriert. Das Front-End gibt kaum Fehlermeldungen zurück. Die Suchfunktion wurde nicht implementiert. Es gibt keine Detailansicht der Artikel. Das grafische Design kann stark verbessert werden.
Das Back-End implementiert weder TLS-Verschlüsselung, noch hasht es die Passwörter der Nutzer. Es wurden keine Sicherheits-Header wie X-Frame, X-XSS-Protection, oder Content Security Policy implementiert. HTTP/2 Push wurde nicht implementiert. Damit würde die Performance erheblich gesteigert. Die Validierung der Eingabedaten prüft nur die Länge des Nutzernams. Die Fehlerbehandlung und das Logging wurden nur grundlegend implementiert.
Der Webshop kann um einige Funktionen erweitert werden. Durch Data-Mining könnten Rückschlüsse vom Kaufverhalten der Nutzer gezogen werden. Der Traffic der Seite könnte analysiert werden, um Fehler zu erkennen und Umschlagraten zu erhöhen. Ein Forum, FAQ-Bereich, oder ein Service um Fehler zu melden, sind nicht vorhanden. Der Nutzer hat keine Möglichkeit Währungen auf der Seite zu tauschen. Bisher muss er vor dem Einkauf Zcash kaufen und es an die Zcash-Adresse des Webshops schicken.

5.2 Politischer Aspekt

Kryptowährungen sind neu und schwer kontrollierbar. Transaktionen können nicht, oder nur schwer realen Personen zugeordnet werden. Die meisten dieser Währungen benötigen kein zentrales Verwaltungsorgan. Eine korrekte Versteuerung dieser Währungen ist damit kaum möglich. Steuern für Einnahmen durch Mining müssen jedoch gezahlt werden. Das Gesetz schreibt vor, Steuern in der Landeswährung zu zahlen. Solange dieses Gesetz gilt, können sich Krypowährungen nicht von Fiat-Währungen lösen. Ein Bürger muss seine Kryptowährung vor der Versteuerung immer erst in eine Fiat-Währung umtauschen. Gesetze für die Versteuerung durch Kryptowährungen sind damit unbedingt nötig. Nach der Meinung des Autors stellt dies eine unüberwindliche Hürde dar.

5.3 Soziologischer Aspekt

Kryptowährungen werden in den Medien fast ausschließlich mit Internetkriminalität in Verbindung gebracht. Die starken Kursschwankungen ziehen Späkulanten an und schrecken Nutzer ab.
Zudem sind Kryptowährungen nicht einfach zugänglich. Diese Technologie unterscheidet sich stark von anderen Währungen. Man kann Transaktionen nicht einfach Rückgängig machen. Der Verlust des privaten Schlüssels führt zum Verlust der Währung. Ein Dieb ist nicht, oder schwer zurückverfolgbar. Zudem kann ein Diebstahl nicht zwingend nachgewiesen werden. Diese Währungen können nicht physisch in die Hand genommen werden.
Dabei erkennt breite Masse die Vorteile dieser Währungen nicht. Viele Menschen denken, diese Währungen hätten keine Existenzgrundlage. Sicher sind die ausschließlich digital, doch damit unterscheiden sie sich kaum von heutigen Fiat-Währungen. Zudem können Kryptowährungen keine In- oder Deflation erleiden. Dagegen können Fiat-Währungen beliebig auf- und entwertet werden. Kryptowährungen hängen von der Sicherheit des privaten Schlüssels ab. Die Sicherheit einer Fiat-Währungen liegt bei dessen Verwaltungsorgan und dem Staat. Diese können Geld nach belieben einfrieren und abziehen. Das Vertrauen in Fiat-Währungen ist irrational. Dieses Geld führte bisher immer zu Wirtschaftskrisen. Staaten haben schon häufig Währungen ersetzt und entwertet. Es besteht eine geringe Wahrscheinlichkeit des Verlusts von Kryptowährung durch Diebstahl. Diebstahl von Fiat-Währung durch den Staat ist gewiss.

Literaturverzeichnis

[1] Shiju Varghese, Prateek Baheti: *Web Development with Go: Building Scalable Web Apps and RESTful Services,* Apress Media, 2015

[2] Brian Ketelsen, Erik St. Martin: *Go in Action,* Manning Publications, 2015

[3] Sau Sheong Chang: *Go Web Programming,* Manning Publications, 2016

[4] Ben Frain: *Responsive Web Design with HTML5 and CSS3 Second Edition,* Packt Publishing, 2015

[5] Vishal Rana, Nitin Rana „Echo - High performance, minimalist Go web framework Guide“, `https://echo.labstack.com/guide`, 12.03.2017

[6] Vishal Rana, Nitin Rana „Echo - High performance, minimalist Go web framework Middleware“, `https://echo.labstack.com/middleware`, 06.06.2017

[7] Vishal Rana, Nitin Rana „Echo - High performance, minimalist Go web framework Cookbook“, `https://echo.labstack.com/cookbook`, 06.06.2017

[8] „GoDoc - echo.v3“, `https://godoc.org/gopkg.in/labstack/echo.v3`, 06.06.2017

[9] „Polymer Project 2.0 - Custom elements“, `https://www.polymer-project.org/2.0/docs/devguide/custom-elements`, 06.06.2017

[10] „Polymer Project 2.0 - Define an element“, `https://www.polymer-project.org/2.0/docs/devguide/registering-elements`, 06.06.2017

[11] „Polymer Project 2.0 - Declare properties“, `https://www.polymer-project.org/2.0/docs/devguide/properties`, 06.06.2017

[12] „Polymer Project 2.0 - Shadow DOM concepts“, `https://www.polymer-project.org/2.0/docs/devguide/shadow-dom`, 06.06.2017

[13] „Polymer Project 2.0 - Handle and fire events“, `https://www.polymer-project.org/2.0/docs/devguide/events`, 06.06.2017

[14] „Polymer Project 2.0 - Data system concepts“, `https://www.polymer-project.org/2.0/docs/devguide/data-system`, 06.06.2017

[15] „Polymer Project 2.0 - Data binding helper elements“, https://www.polymer-project.org/2.0/docs/devguide/templates, 06.06.2017

[16] „Polymer Iconset Generator“, https://poly-icon.appspot.com, 06.06.2017

[17] „Introduction“, https://www.webcomponents.org/introduction, 06.06.2017

[18] „paper-button“, https://www.webcomponents.org/element/PolymerElements/paper-button, 06.06.2017

[19] „app-layout“, https://www.webcomponents.org/element/PolymerElements/app-layout, 06.06.2017

[20] „paper-item“, https://www.webcomponents.org/element/PolymerElements/paper-item, 06.06.2017

[21] „paper-input“, https://www.webcomponents.org/element/PolymerElements/paper-input, 06.06.2017

[22] „iron-icon“, https://www.webcomponents.org/element/PolymerElements/iron-icon, 06.06.2017

[23] „iron-image“, https://www.webcomponents.org/element/PolymerElements/iron-image, 06.06.2017

[24] „iron-iconset-svg“, https://www.webcomponents.org/element/PolymerElements/iron-iconset-svg, 06.06.2017

[25] „GoDoc - httpexpect“, https://godoc.org/github.com/gavv/httpexpect, 06.06.2017

[26] „GoDoc - mgo.v2“, https://godoc.org/gopkg.in/mgo.v2, 06.06.2017

[27] „Query Documents“, https://docs.mongodb.com/manual/tutorial/query-documents/, 06.06.2017

[28] „Update Documents“, https://docs.mongodb.com/manual/tutorial/update-documents/, 06.06.2017

[29] Matt Silverlock „Generating Secure Random Numbers Using crypto/rand “, https://elithrar.github.io/article/generating-secure-random-numbers-crypto-rand/, 09.05.2017

[30] „Polymer tools“, https://www.polymer-project.org/2.0/docs/tools/polymer-json, 06.06.2017

[31] „IndexedDB Promised“, https://www.npmjs.com/package/idb, 19.06.2017

[32] „How Zcash works“, https://z.cash/technology/index.html#how-it-works, 19.06.2017

[33] „Original Bitcoin client/API calls list“, https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list, 19.06.2017

[34] „Zcash Payment API“, https://github.com/zcash/zcash/blob/master/doc/payment-api.md, 19.06.2017

[35] Matt Gaunt „Service Workers: an Introduction “, https://developers.google.com/web/fundamentals/getting-started/primers/service-workers, 25.06.2017

[36] Eric Bidelman „Shadow DOM v1: Self-Contained Web Components“, https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom, 25.06.2017

[37] „Polymer 2 “, https://www.polymer-project.org/, 20.08.2017

[38] „Go “, https://golang.org/, 20.08.2017

[39] „zk-SNARKs “, https://z.cash/technology/zksnarks.html, 20.08.2017

[40] „Material Design “, https://material.io/, 20.08.2017

[41] „Shadow DOM “, https://www.w3.org/TR/shadow-dom/, 20.08.2017

[42] „Service Worker “, https://developers.google.com/web/fundamentals/getting-started/primers/service-workers, 20.08.2017

[43] „Indexed Database API “, https://www.w3.org/TR/IndexedDB/, 20.08.2017

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 20. August 2017